AFRL-IF-RS-TR-1998-34
Final Technical Report
April 1998

ATLANTIS: AN OPEN ARCHITECTURE FOR SYNERGY OF PROCESS-CENTERED ENVIRONMENTS AND COMPUTER-SUPPORTED COOPERATIVE WORK

Columbia University

Sponsored by
Advanced Research Projects Agency
ARPA Order No. B128

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

AFRL-IF-RS-TR-1998-34 has been reviewed and is approved for publication.

APPROVED: [illegible]
JAMES R. MILLIGAN
Project Engineer

FOR THE DIRECTOR: [illegible]
NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

ATLANTIS: AN OPEN ARCHITECTURE FOR SYNERGY OF PROCESS-CENTERED ENVIRONMENTS AND COMPUTER-SUPPORT COOPERATIVE WORK

Gail E. Kaiser

Contractor:	Columbia University
Contract Number:	F30602-94-C-0197
Effective Date of Contract:	30 June 1994
Contract Expiration Date:	30 September 1997
Program Code Number:	7E20
Short Title of Work:	ATLANTIS: COLUMBIA SUPPORT
Period of Work Covered:	Jun 94 - Sep 97
Principal Investigator:	Gail E. Kaiser
Phone:	(212) 939-7084
AFRL Project Engineer:	James R. Milligan
Phone:	(315) 330-2054

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by James R. Milligan, AFRL/IFTD, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** April 1998	**3. REPORT TYPE AND DATES COVERED** Final Jun 94 - Sep 97

4. TITLE AND SUBTITLE ATLANTIS: AN OPEN ARCHITECTURE FOR SYNERGY OF PROCESS-CENTERED ENVIRONMENTS AND COMPUTER-SUPPORTED COOPERATIVE WORK	**5. FUNDING NUMBERS** C - F30602-94-C-0197 PE - 62301E PR - B128 TA - 01 WU - 01
6. AUTHOR(S) Gail E. Kaiser	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Columbia University Department of Computer Science 1214 Amsterdam Avenue, Mail Code 0401 New York NY 10027	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Advanced Research Projects Agency 3701 North Fairfax Drive Arlington VA 22203-1714 / AFRL/IFTD 525 Brooks Road Rome NY 13441-4505	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-1998-34

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: James R. Milligan/IFTD/(315) 330-2054

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

This report summarizes the work performed under a DARPA-sponsored effort performed by Columbia University which focused on the development and integration of advanced software engineering environment capabilities for computer-supported cooperative work, rule-based process modeling and execution, and distributed transactions management.

14. SUBJECT TERMS Process-Centered Environments, Groupware, Software Engineering, CSCW, Transaction Management	**15. NUMBER OF PAGES** 272
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

List of Figures

Preface

The *Atlantis* project at Columbia University investigated componentization of workflow modeling and execution systems, particularly synergy of process-centered environment and computer-supported cooperative work components. The components are intended to be interoperable with legacy and off-the-shelf tools and frameworks and indicate requirements on future systems, for a concrete transition path.

Subject Terms: Computer-Supported Cooperative Work, Distributed Computing, Object-Oriented Database, Software Development Environments, Software Process, Transaction Management, Workflow

This document describes work sponsored in part by Defense Advanced Research Project Agency under DARPA Order B128 monitored by Rome Lab F30602-94-C-0197.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US government or DARPA.

Chapter 1

Overview

1.1 Project Summary

Atlantis was originally formulated as a consortium consisting of the Programming Systems Lab at Columbia University, the Advanced Collaborative Systems Lab at the University of Illinois at Urbana-Champaign and the US Applied Research Lab of Bull HN Information Systems. However, the Illinois work was later partially conducted at the Distributed Systems Technology Centre at University of Queensland, Australia; and ARL shut down, their DARPA contract was cancelled. The two academic groups had been working on their own diverse workflow-related projects for over a decade prior to forming *Atlantis*. They initiated cross-licensing with each other and with Bull to evaluate the prospects for integrating their technologies, culminating in a plan to investigate several important, practical problems not previously being pursued:

- **Groupware**. The main theme was to integrate *human/human collaboration*, studied in the computer-supported cooperative work (CSCW) community, with *tool/tool integration*, the forte of the software engineering community. The Illinois researchers had addressed process primarily from a CSCW perspective, e.g., Bull used their framework to develop a system for directing human behavior during document inspections. The Columbia lab had been working on multi-user process-centered environments (PCEs), particularly enforcement of process constraints and automation of tool invocations to satisfy prerequisites and fulfill consequences of process steps. The open architecture for workflow systems drew from both lines of research, with the results originally intended to be transitioned by Bull into potential products.

- **Process transition** from current computer-aided software engineering (CASE) and tool integration technologies to PCEs by developing an *external* process server component that enacts project-specific process definitions. The resulting *open architecture* supports mediation between such a server and applications, to minimize or avoid changes to pre-existing systems. The process server component provides a rule-based "process assembly language" into which the user organization's choice of process modeling formalism (e.g., Petri nets, grammars, task graphs) can be translated, to be executed by the corresponding "process virtual machine", thus lowering the barrier to adoption.

- **Collaboration transition** from current database transactions and "check-out" technology to collaborative workflow environments by developing an *external* cooperative transaction manager. It is widely agreed that the classical transaction model is inappropriate for long-duration, interactive and/or cooperative activities, but there is no consensus on the numerous extended transaction models that have been proposed, and it appears that there will never be a consensus: different models are needed for different applications. Thus the transaction-management component supplies primitives for defining project-specific concurrency control policies, analogous to process modeling. The *open architecture* enables mediation between the transaction manager and pre-existing systems, mapping task units to transaction-like constructs.

- **Geographical distribution**. Industrial-scale software development increasingly takes place outside the boundaries of a local area network, often spread across regions and/or independent organizations, on an intranet/extranet or the Internet. Collaborating subcontractors may guard their own proprietary processes and tools, while sharing data subject to security constraints, so a model for "cooperating software processes" is needed. The *open architecture* extends workflow management and execution technology to *interoperability* among autonomously defined processes, with a wide area network-capable PCE infrastructure.

1.2 Accomplishments

The main deliverables for the project overall were:

- Definition and realization (Oz) of the Treaty/Summit model for interoperability among autonomously defined processes and an infrastructure for Internet-wide distribution.
- Development of a coordination modeling language and interpreter (CORD), and several case studies interfacing an external concurrency control component (Pern) to existing systems (including the commercial ProcessWEAVER).
- Definition of a process assembly language and corresponding process virtual machine (Amber), and several case studies translating other workflow definition formalisms (including commercial StateMate and ProcessWEAVER notations) and interfacing Marvel or Amber to existing systems (including ProcessWEAVER and Bull's Scrutiny).
- Construction of an object management system component (Darkover) from Oz's native OMS, and replacement of the "hidden" file system (for coarse-grained data) with a more efficient subsystem.

1.2.1 FY94-FY95

The *Atlantis* project at Columbia University developed an "International Alliance" metaphor for interoperability and collaboration among diverse workflow processes. The processes may be specified autonomously by independent organizations, with interoperation added-on if and when needed while the processes are in progress, or alternatively may be specified top-down as interacting subprocesses when collaboration is planned in advance. A Treaty is a mechanism for agreeing on small to large degrees of collaboration among processes (termed "sites"), at the level of one or more workflow steps. Any two or more sites may separately choose whether or not to participate in a Treaty, and any site can decide to cancel its agreement. Treaties need not be symmetric; e.g., one organization may agree to perform certain operations on another organization's data without permitting that other organization to apply those operations to its own data.

A Summit is an execution of any process step involving data from multiple sites; all of these sites must have previously agreed to the corresponding Treaty. A Summit may involve multiple processes executing on the same machine, on different machines connected by a local area network, or on collections of machines arbitrarily dispersed over a wide area network such as the Internet. Any prerequisites to a Summit process step are requested from each local site, to be satisfied according to its own private process; a Summit step cannot continue if any affected site does not fulfill the relevant prerequisite(s). Similarly, each site is notified of the consequences of a Summit, and may (or may not) carry out corresponding actions according to its own private process. A Summit can potentially consist of a sequence of coordinated steps. The basic Treaty/Summit approach applies in principle to any process modeling and execution technology, but to date has been implemented only for Oz.

The Oz system provides an infrastructure for registration and communication of sites that could potentially collaborate. Oz currently implements the International Alliance metaphor for a rule-based process engine, where each process step is specified by a condition-action-effect rule, and execution of tasks consisting of multiple process steps is implemented by backward and forward chaining among rules. A Treaty corresponds to a selected subset of the rules, and requires a common subschema describing the structure of relevant product and process data. A Summit involves backward chaining at each local site to satisfy the condition and forward chaining at each local site to carry out the implications of the effect, with one site coordinating the execution of the Summit rule. This rule might invoke a conventional tool, with the tool's user interface supplied either to the user who invoked the Summit in the first place or to an alternative user from a specified list, or instead a groupware tool with simultaneous delegation to multiple users. Rule chains may also be formed among Summit rules when the collaboration consists of several directly linked phases.

The Columbia group also achieved something that many believed "couldn't be done": adding transaction management to legacy systems. A mediator-based architecture allow system builders to add concurrency control to existing database applications, environment frameworks, and workflow execution systems – in many cases without changing the existing software. The Pern external concurreny control component requires the application to have some notion of grouping into task-like units (which are mapped to transactions), some mechanism for wrapping calls or messages to Pern around data accesses (for locking), and some capability for undoing or rolling back updates (for recovery). Many systems do not directly provide such "hooks", but back doors into a system's facilities can sometimes be found; in the worst case, recovery can be achieved by Pern submitting a series of new update that happen to restore the previous values of the affected entities. Only conventional atomic and serializable transactions were supported initially.

Columbia's predecessor Marvel 3.x system, which supported a single process model per environment resident on a local area net, was licensed to over 40 institutions including a dozen companies and Rome Lab. An early version of the Oz process-centered environment, supporting logical distribution and multi-process interoperability, was delivered to DARPA with complete user and administrator manuals. Oz was in daily use for over two years in daily internal use for both software development and technical writing, prior to the deployment of the successor OzWeb system.

1.2.2 FY96

Database management systems (DBMSs) are increasingly used for advanced application domains, such as software development environments, network management, workflow management systems, CAD/CAM, and managed healthcare, where the standard correctness model of transaction bserializability is often too restrictive. The *Atlantis* project at Columbia introduced the notion of a Concurrency Control Language (CCL) that allows a database application designer to specify concurrency control *policies* to tailor the behavior of a transaction manager: a well-crafted set of policies defines an extended transaction model. The necessary semantic information required by the CCL run-time interpreter is extracted from a task manager, a (logical) module by definition included in all advanced applications, which stores task models that encode the semantic information about the transaction-like units submitted to the DBMS. Columbia designed a rule-based (condition-action) CCL, called CORD, and implemented a run-time interpreter that can be hooked to a conventional transaction manager to implement the sophisticated concurrency control required by advanced database applications. They developed an architecture for systems based on CORD, integrated the CORD interpreter with their own Pern transaction manager component and with the Exodus storage manager from the University of Wisconsin, and implemented the well-known Altruistic Locking and Epsilon Serializability extended transaction models as samples.

Columbia also developed a process server component, called Amber. New systems could be constructed around the component, or existing non-process environment architectures enhanced with value-added process enactment services (as in Columbia's integration with Field from Brown University). Synergistic integration with existing process engines enables a degree of heterogeneity (as in Columbia's coupling with their mockup of TeamWare from the University of California at Irvine), and previous-generation process enactment facilities in existing environments could be replaced (as Columbia did with their own Oz system and with the ProcessWEAVER product from Cap Gemini).

Amber also supports translation of higher-level process modeling formalisms, e.g., Petri nets, grammars and graphs, into its rule-based "process assembly language" for enactment, and permits addition of formalism-specific support to the process engine via an extension and parameterization mechanism. Thus the same process engine can support many different process modeling paradigms. Columbia developed translators for ProcessWEAVER's Cooperative Procedures (Petri nets), Bill Riddle's Activity Structures (concurrent regular expressions), and StateMate's statecharts (finite state automata). Integration of Amber as an external component into some foreign system, and extension/parameterization of its process syntax and semantics, are both achieved through callbacks to a mediator consisting of special-purpose "glue" code. The Amber version of Oz, replacing its native process engine, was used for all software development by the Columbia Atlantis group.

"Black Box" enveloping technology expects the tool integrator to write a special-purpose script to handle the details of interfacing between each COTS (or GOTS) tool and the environment framework. Generally, the complete set of arguments from the environment's data repository is supplied to the tool at its invocation and any results are gathered only when the tool terminates, so tool execution is encapsulated within an individual task. This does not work very well for: Incremental tools that request parameters and/or return (partial) results in the middle of their execution, e.g., multi-buffer editors and interactive debuggers; Interpretive tools that maintain a complex in-memory state reflecting progress through a series of operations, e.g., "Knowledge-Based Software Assistants"; Collaborative tools that support direct interaction among multiple users, including asynchronous discussion and synchronous conferencing.

To address these concerns, the *Atlantis* project at Columbia introduced a Multi-Tool Protocol (MTP), which enables submission of multiple tasks, either serially or concurrently, to the same executing tool instance on behalf of the same or different users. Single-user tools can thereby be converted to a "floor-passing" form of groupware, and a user can even send a running task to another user for assistance; these facilities assume X11 Windows. MTP also addresses multiple platforms: transmitting tool invocations to machines other than were the user is logged in, e.g., when the tool runs only on particular machine architecture or is licensed only for a specific host. Columbia implemented MTP as part of their Oz process-centered environment.

Componentization involves restructuring a stovepipe system into components that could potentially be reused in other systems, and/or re-engineering an old system to permit replacement of native code with new components. Columbia's *Atlantis* project developed a series of two processes, OzMarvel (running on top of the earlier Marvel process-centered environment) and EmeraldCity (on top of the Oz process-centered environment, Marvel's successor). Each was intended to support both aspects of componentization, but EmeraldCity addressed one requirement not fully understood when OzMarvel was developed: process support for co-existing components designed as alternatives to each other in plug-n-play style. Columbia used OzMarvel to replace Oz's native transaction management subsystem with Pern, and used EmeraldCity both to replace Oz's native object management system with an object-oriented database management system component (called Darkover) and to replace Oz's native process engine with Amber.

1.3 FY97

The *Atlantis* project at Columbia externally released a fully documented and robust version of the Oz process-centered environment framework, and the components described above, ported to Solaris 2.5 (earlier versions ran on SunOS 4.1.3). There are already several "alpha" sites: the University of Massachusetts at Amherst, North Carolina State University, the University of West Virginia, Brown University, the University of California at Irvine, Lockheed Martin and the Naval Research Laboratory.

Columbia completed a "proof-of-concept" World Wide Web browser user interface client to Oz. They developed a general method for accessing legacy client/server applications from standard WWW browsers. The main effort is to modify an existing client for the system to perform HTTP proxy server duties. Web browser users then simply configure their browsers to use this proxy, and thereafter can access the target system via specially encoded URLs that the proxy intercepts and sends to the existing server. The Web-based browser user interface to Oz is just one example.

Columbia developed mechanisms to structure information so that the view of the World Wide Web, both within and across Web pages, is dynamically customizable. They investigated an architecture that integrates environment data repositories, such as their Darkover object management system, with WWW to organize such dynamic structures. Different users, or the same user at different times, could have different views of the Web. Their architecture potentially provides high flexibility for a wide variety of applications, nothing is specific to software development environments.

Chapter 2

Decentralized Process Interoperability

Abstract

A process-centered software engineering environment (PSEE) enables to model, evolve, and enact the process of software development and maintenance. This chapter addresses the problem of process-interoperability among decentralized and autonomous PSEEs by presenting the generic International Alliance model, which consists of two elements, namely Treaty and Summit. The Treaty abstraction allows pairwise peer-peer definition of multi-site shared sub-processes that are integrated inside each of the participating sites, while retaining the definition- and evolution-autonomy of non-shared local sub-processes. Summits are the execution abstraction for Treaty-defined sub-processes. They enact Treaty sub-processes in multiple sites by successively alternating between shared and private execution modes: the former is used for the synchronous execution of the shared activities, and the latter is used for the autonomous execution of any private subtasks emanating from the shared activities. We describe the realization of the models in the Oz multi-site PSEE and evaluate the models and system based on experience gained from using Oz for production purposes. We also consider the application of the model to Petri net-based and grammar-based PSEEs.

2.1 Introduction

As software systems become more complex and larger in scale, their development and maintenance requires more people with various skills, often organized into groups. The decomposition into groups can be characterized by the level of intra-group vs. inter-group heterogeneity. For example, a project may be composed of separate teams for requirements elicitation, functional specification, design, coding, testing, documentation, and maintenance. This decomposition exhibits high intra-group homogeneity and inter-group heterogeneity. Alternatively, a project may be decomposed into teams that are each responsible for full development of a distinct component of the system, exhibiting intra-group heterogeneity. Another characteristic of the project organization is whether it is formed top-down or bottom-up. An example of the latter is when multiple independent organizations team up (perhaps for a limited period of time) to develop a system. Finally, project personnel may be divided into, or made up of pre-existing, physically dispersed teams (or even individuals, e.g. telecommuting from home), a scenario that becomes more frequent with the advances in networking technologies.

Although the various decompositions have their own specific requirements, a common desirable property in a multi-team development is to allow some degree of operational as well as managerial team *autonomy*. For example, it may be desirable to allow teams to use their own set of software tools and hardware, their own private files or databases, and their own development policies and workflow, or process. Furthermore, when the teams belong to different organizations, autonomy and privacy are "hard" constraints that cannot be compromised or a priori restricted. At the same time, the autonomous teams need to *collaborate* in order to develop the product. For example, they may need to share tools or employ multi-user tools across teams, they may need to exchange and/or share files and other data, and they may need to agree on some common policies and workflow, at least for the parts of the work that involve collaboration.

The concept of system-interoperability, which has been largely motivated by the emerging globalization of computing, has been increasingly gaining popularity in various domains such as workflow interoperability for business process re-engineering [81], multi-database interoperability [37], and general client-server interoperability [214, 149]).

In this chapter we explore interoperability in the context of process-centered software engineering environments (PSEEs). PSEEs are systems that support large scale software development by providing: (1) mechanisms and notations for explicitly *modeling* the process of development and maintenance of software, including task definitions, control integration such as global task ordering and local constraints on their activation, tool integration, data modeling and integration, and user modeling; and (2) mechanisms for *enacting* the modeled process by the PSEEs process-engine, where forms of enactment include process automation (e.g., Marvel [124]), consistency (e.g., CLF [185]), monitoring (e.g., Provence [140]), enforcement (e.g., Darwin [162]), and guidance (e.g., Merlin [201]).

Thus, the PSEE-interoperability problem is to balance autonomy and collaboration among multiple *processes*, both in the modeling and the enactment phases, as a basis for collaboration among multiple groups.

2.1.1 Requirements and Scope

Decentralized PSEEs

We deliberately use the term decentralization as opposed to distribution to emphasize that our focus in this work is on interconnecting environments that are independent and loosely-coupled, both physically and logically. This is in contrast to a "classical" distributed system in which a single and homogeneous logical perspective is given to its applications but is physically distributed into multiple computing units. Note that PSEE distribution (in the "classical" meaning) is a form of "vertical" scale-up, in that it allows for more users to work, but under the same process and typically with some bounded physical distance (typically a local-area network). Here we address mainly "horizontal" scale-up, where the number of users per group sharing the same process may not grow much (and in fact may degenerate to a single user), but the number of groups may be arbitrarily large, each group with its own private process and data but collaborating in a concerted effort with other groups.

Another aspect that is derived from decentralization is *Independent Operation and Self-Containment*. This means that a sub-environment (henceforth SubEnv) should be able to behave as a complete environment by itself when not collaborating with any other SubEnvs, and SubEnvs must be able to operate concurrently and independently, except when their processes explicitly collaborate. The most fundamental implication of this requirement is a "share-nothing" architecture. That is, no multi-site service, mechanism, or data in the environment can be centralized or physically shared and all interaction should be based solely on message passing.

Decentralization also implies that A multi-site PSEE should impose minimum overhead on the operation of local work in SubEnvs. The underlying assumption is that most of the work done by a SubEnv is local to that SubEnv, and therefore each SubEnv should still be optimized towards local work.

Finally, we make the distinction between inter- vs. intra- process coordination. The latter is concerned with coordinating concurrent activities that might violate the consistency of the project database, assuming that all participants use the same process, the same schema, and most importantly, share the same centralized, project database (see, for example, [14]). In contrast, we focus in this chapter on collaboration between users or teams with different processes, different schemas, and different project databases.

PSEE Autonomy

Each local SubEnv should have complete control over its process, tools and data, while allowing access by remote SubEnvs under restrictions that are solely determined by the local SubEnv. Access to a SubEnv has two perspectives: access to the local artifacts owned by the SubEnv through some process interface; and access to, and interaction with tools and actual process tasks. Autonomy constraints imply that a SubEnv's data, tools and process are by default private, and some work has to be done to allow sharing and remote use. Moreover, once defined, sharing should be restricted to the minimum degree necessary for interoperability.

Process Heterogeneity

Heterogeneity in software systems in general (and PSEEs in particular) can be classified into four levels: the operating system, the runtime support (PSEE engine), the front-end language (Process Modeling Language (PML)), and the applications (specific processes). For example, a multi-PSEE can support heterogeneous processes written in the same PML (application heterogeneity), it can support heterogeneous PMLs but still require the same underlying (multi-lingual, in this case) process engine (language heterogeneity), or it can support interoperability across heterogeneous PSEE engines (system heterogeneity). Support for heterogeneity is, in general, an extremely difficult problem, particularly in the context of decentralization. This chapter explores a limited aspect of heterogeneity by fixing the system and language levels (although not restricting to *a* particular system or language) and supporting heterogeneity at the process model level. This is in contrast to ProcessWall [98] for example, which focuses on language heterogeneity (see Section 2.6).

Allowing Pre-Existing Processes

In a typical top-down approach a system (process) is decomposed into sub-systems (sub-processes), usually by a global authority which dictates where and how the different parts of the system, both control and data, will be defined and executed. In contrast, the bottom-up approach, which is closely associated with decentralized systems, does not assume a global authority, and multi-site applications are constructed between the possibly pre-existing local (sub)systems, thereby avoiding the need to have any a priori knowledge of the "neighboring" subsystems before the time of construction. Our focus here is on the more decentralized bottom-up construction of multi-PSEE processes. This is in contrast to most other distributed PSEEs which decompose a single process in a top-down fashion into sub-processes with predefined and coordinated interfaces (as done in ProcessWEAVER [67], see Section 2.6). Notice that we do not exclude support for top-down methodology; we do not, however, enforce it.

Thus, it should be possible for pre-existing SubEnvs to "join" an on-going multi-site environment or to form a new one with minimal configuration overhead. Similarly, a "split" of a SubEnv from its currently configured multi-site environment should be supported.

The rest of this chapter is organized as follows: Section 2.2 presents the interoperability model for definition and enactment of multi-site activities; Section 2.3 discusses an actual implementation of the model in the Oz rule-based multi-site PSEE; Section 2.4 outlines the application of the model to other non-rule-based PMLs; Section 2.5 evaluates the research based on experience gained by using Oz in a production environment; Section 2.6 compares to related work; and Section 2.7 summarizes the contributions of this research and outlines future directions.

In earlier work [27] we introduced a preliminary version of the model and its implementation, focusing on enactment. Our book [21] presented a revised, comprehensive and formalized model with detailed coverage of both the definition and enactment aspects, and describes a mature implementation. This chapter of the report abridges the book, and adds new material in two areas: Section 2.5 is entirely new. It describes anecdotal experience and provides statistics on using one Oz environment for production purposes by up to 14 users over approximately 8 months (to date), and (re)evaluates the interoperability model based on this experience. This and other retrospective led to abstraction of local process evolution and dynamic Treaty verification out of the Oz realization and its generalization and reformulation as part of the International Alliance model in Section 2.2.2.

2.2 The Process Interoperability Model

At a high-level, our approach taken to meet the challenges described above is to exploit the fact that process models are encoded in a formal notation, and use it as a basis for formally modeling *interoperability* among process models. Furthermore, we extend the concept of process enactment to encompass enactment support for the actual multi-process activities that enable collaboration between the sites, in addition to the conventional single-site execution.

We begin with definitions of terms and a formalization of concepts that are used in the rest of the chapter.

2.2.1 Basic Concepts and Definitions

2.2.1.1 General Process Terminology

As stated earlier, a process model defines a project-specific process and is encoded in some process modeling language (PML), and a process-centered software engineering environment (PSEE) is a system in which processes are modeled and enacted. A process model can be *instantiated* when it gets bound with real data artifacts, tools, users, and any other system bindings which are required by the PSEE. An instantiated environment is an executable process model. For brevity, we shall call an instantiated environment simply an *environment* (or SubEnv in the context of multi-site PSEEs). This term should not be confused with the term PSEE, which refers to the system on which (instantiated) environments run.

Note that the same process model can be instantiated in multiple environment instances. At some point during its enactment, the process model of an environment might need to be changed, e.g., because of feedback from the environment and/or new requirements, in which case it is *evolved*, i.e., its persistent process and product states are upgraded to comply with the new process definition.

We can identify a generic three-level context hierarchy in process models. A particular PML may have more or fewer levels, but we assume that there is some mapping into these core levels:

1. *Activity* — This is the PSEE's interface to actual tools, including input/output data bindings, user bindings (i.e., who should execute the tool if it is interactive), and machine binding (i.e., on what machine should the tool execute).

2. *Process-step* — This level encapsulates an activity with local prerequisites and immediate consequences (if any) of the tool invocation, as imposed by the process. For example, in the FUNSOFT Petri net based PML [90] a process step corresponds to a transition along with its (optionally) attached predicates; in the Articulator task graphs [161] this level corresponds to a node with its predecessor and successor edges; and in rule-based PMLs, a process step is represented by a rule with pre- and post-conditions. The process-step level may also supply the mechanism to interface among multiple activities in a process. For instance, in rule-based PMLs, a post-condition of one rule is matched against a pre-condition of another rule to determine possible chaining; similarly, the firing of a Petri net transition can enable another transition.

3. *Task* — A set of logically related process steps that represent a coherent process fragment. Depending on the specific PML and PSEE (1) there usually are some ordering constraints among the activities or process steps of a task; (2) parts of a task might possibly be inferred dynamically, emanating from an entry activity or process step selected by the user; and (3) depending on the subtasks, a task might be partially carried out automatically by the PSEE on behalf of the user, usually by triggering the inferred activities or steps. The task level may be explicitly defined in the PML through a special notation, or may be implicitly defined through the local prerequisites/consequences in the process-step level, or both.

2.2.1.2 A Multi-User, Single-Process PSEE

An (instantiated) environment E is defined as a quintuple

$$E = < U, T, S, D, P >$$

where:

- U is a set of users using the environment. No built-in roles or hierarchies are assumed to be attached to users, except for the concept of an environment *administrator*, who defines and can modify each of the elements in E (analogous to the role of a database administrator).

- T is a set of *tools* being used in the environment. The tools can be off-the-shelf, or customized to work in the PSEE, but in either case the PSEE is assumed to have means to invoke those tools with process activities.

- S is a *schema* sub-language representing data types for modeling the process and product data. S could be part of an external database that is separate from the PML (as in SPADE [10]) or it could be part of the PML (as in Marvel [124]). In addition, the process data could be kept separately from the product data (which may reside in the native file system). In PMLs with no data modeling at all (e.g., Synervision [109]) this element degenerates to the empty language and all data elements are considered to belong to the single "universal" class.

- D is a *database* for storing the persistent objects, each belonging to a certain type (or class) from S.

- P is a set of activities/steps/tasks and their inter-relationships, which together comprise the *process* model. They can be invoked either manually by human end-users, or automatically by the process engine. Each activity encapsulates a tool from T, with formal parameters from S, and actual parameters from D. An activity is not required to be bound to specific users (or roles) from U, although such a requirement can be imposed by a specific implementation or a specific process definition.

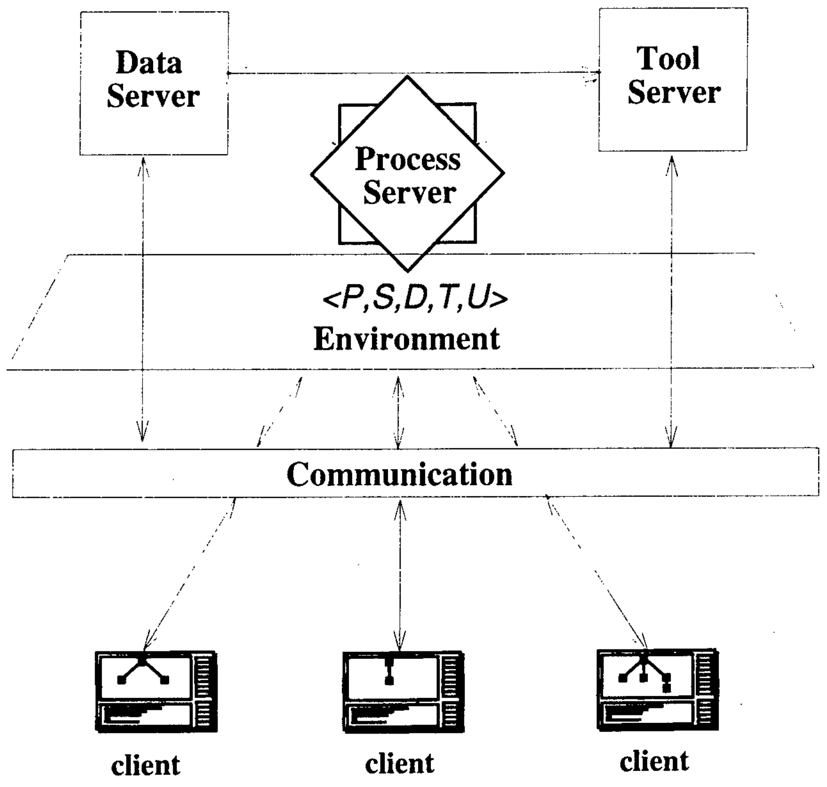


Figure 2.1: A Generic Multi-User Single-Process Environment

Based on the above definitions and requirements, a high-level view of a single-process PSEE with an instantiated environment is depicted in Figure 2.1. It consists of a data server managing the process schema and data, a tool server integrating the project's tools, a process server executing the defined process, a client-user interface, and a communication layer connecting all components. A typical interaction with the PSEE is as follows: an end-user from U initiates a task from P by invoking an activity that encapsulates tool(s) from T, on a set of data arguments from D that belong to classes from S. The process server receives the request, and depending on the specific installed process and other ongoing activities, determines what to do before, during and after the requested activity, involving the data and tool servers, which can also interact directly with the client.

2.2.1.3 A Multi-Group, Multi-Process PSEE

A multi-process decentralized environment is formally defined as:

$$\{E_i\}\, i = 1 \ldots n$$

where each E_i is a single-process environment as defined above, maintaining its own data repository, tools, and process model. While the data is disjoint, it must nonetheless be accessible by remote SubEnvs in order to enable process-interoperability. Thus, we assume that the underlying PSEE has the necessary mechanisms to reference and bind remote data objects to local activities. Driven by autonomy requirements, however, the data in each SubEnv is private by default, and is said to be "owned" by its local process. Thus, access to both process and product data cannot be made from a remote process without prior authorization from the owner process.

The high-level architectural view of a generic decentralized PSEE with a three-site decentralized environment is depicted in Figure 2.2. Each local SubEnv consists, in addition to the single-process components, of an inter-process server, a remote-data server, a remote-tool server and, a connectivity

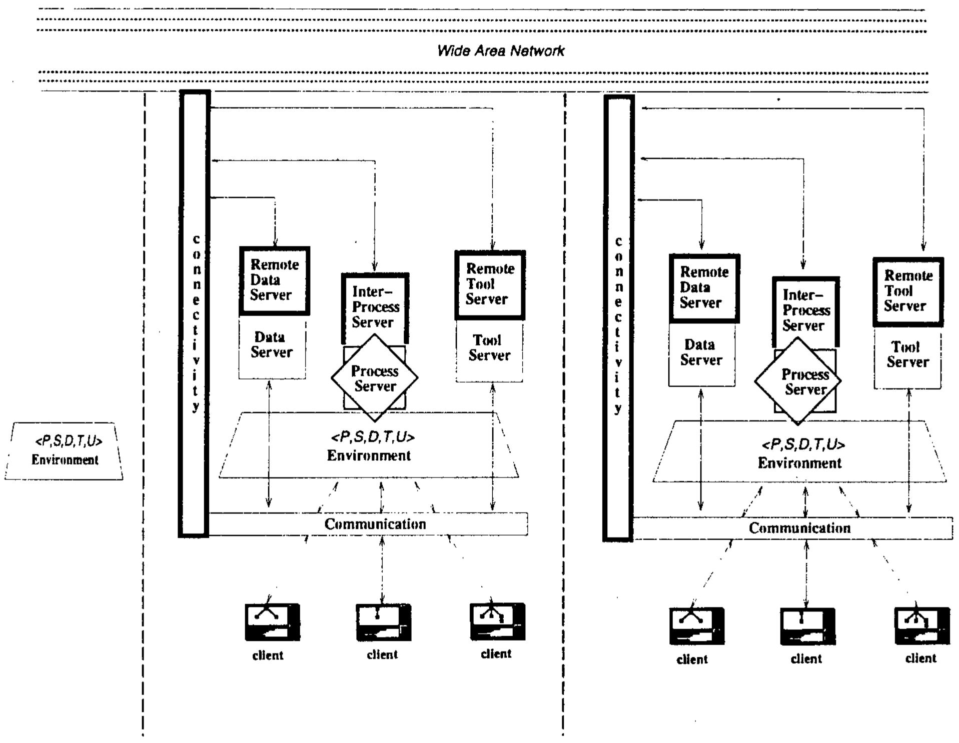


Figure 2.2: A Decentralized Environment

server that enables SubEnvs to connect to, and communicate with, other SubEnvs participating in the same (global) environment. These elements together form the necessary infrastructure support needed for process-interoperability. Notice how the "no sharing" property allows normal operation of some sites when other sites (e.g., the leftmost site in the figure) are inactive or disconnected.

We define a *multi-site activity* as an activity that uses data objects, and optionally users and/or machines from one or more remote sites. Note that the activity (or tool) per-se, need not execute on multiple sites concurrently (as in groupware tools), it can execute at one site into which the remote data objects are transferred. Thus, a given activity may or may not be considered a multi-site activity during different invocations, depending on whether the resources bound to it include remote elements. Multi-site activities are the building blocks of our process-interoperability model.

Finally, referring back to the context-hierarchy described earlier, it is important to note that there is intentionally no fourth level that represents a local process as part of a global process. This reflects our concept of *independent* collaborating (local) processes. While our model provides global infrastructure support to enable interoperability among local processes, it explicitly avoids the need for a global "super" process — although such a process can be implicit.

2.2.2 Defining Process Interoperability: the Treaty

In general, the interoperability model is based on the idea that sites explicitly specify in what ways they are willing to participate in a multi-site operation, and the specifications are loaded into each site's local PSEE to establish all that is needed to enable those interactions. Some intuition to the model may be gained by the "international alliance" metaphor, whereby independent countries sign "treaties" that determine their collaboration but retain full control over their local laws. Once signed, treaties have to be ratified by the local parties, so that the full impact of the treaty is reflected in each country when enacted. In addition, Treaties have to be verified to make sure that they are being carried out as agreed.

2.2.2.1 Treaty Requirements

The following is a set of requirements specific to modeling interoperability, driven by the high-level requirements presented earlier in Section 2.1.1.

1. *Common sub-process* — In order to enable invocation of multi-site activities, there must be a way to define and agree on a common sub-process that would become an integral part of each local process intended to collaborate during that sub-process (but not necessarily by all SubEnvs in a global environment). A common sub-process determines what actions can be taken in the multiple participating SubEnvs. At the very least, the multi-site activities must be commonly specified so that they can be identified during execution. But this "unit of commonality" might also be the process step, or even the task. In any case, this unit represents those process fragments that potentially involve multiple local processes. The decision as to what level (in the context hierarchy) to choose as the unit of commonality depends on the modeling primitives of the specific PML. In a Petri net formalism, for example, the transition (along with its input and output places) seems a natural choice, whereas in rule-based PMLs the rule (process step) is likely to be chosen. In PMLs that support task hierarchies and modularization (e.g., Articulator [161]), a subtask might be the right choice.

   It is important to recognize that the activity portion of a decentralized sub-process need not be executable in every participating SubEnv, e.g., since the encapsulated tool may not be physically available everywhere. Instead, the activity only needs to be executable in <u>one</u> of the SubEnvs intended to collaborate, which would hence always serve as the invoking, or *coordinating* process. This means that common sub-processes are not necessarily reciprocal, in the sense that not all participant SubEnvs have identical process "privileges" on multi-site activities. This issue has direct implications on the model, as will be seen shortly.

2. *Common sub-schema* — This requirement applies mainly to PSEEs with database and schema support. In such PMLs, invocation of multi-site activities (as part of a multi-site common sub-process) requires the involved SubEnvs to share a *common sub-schema*, so that the types of the parameters specified in the invocation are defined in the relevant SubEnvs. For example, if an activity A_1 is invoked from SubEnv E_1 on remote data from E_2, then E_2 must have the proper data types (with possible support for limited type coercion) in its schema and consequently the properly instantiated objects that are required by A_1. Note, however, that a common sub-schema does not necessarily imply that the corresponding data <u>instances</u> are shared — only their types (i.e., their schema) are shared. Defining common data schema and allowing access to data instances are separate concerns which should not be confused or coalesced.

3. *Remote access control* — Following the above argument, there must be a way to define (and subsequently, control) which data instances are allowed to be accessed, in what way, and by which SubEnv. That is, local databases are by default private, consistent with the autonomy requirement, but parts of them can be made accessible for remote access by multi-site activities.

4. *Locality of specifications* — It must be possible for a common sub-process (and the corresponding common sub-schema) to be shared among only some of the local processes of a given global environment, not necessarily all of them. Furthermore, a SubEnv may contain multiple sub-processes, each of which is shared with different subsets of peer SubEnvs. There is usually some portion of each local process that is not shared with any other process (a *private* sub-process). Similarly, it must be possible to specify access to subsets of the data instances to only some but not all participating SubEnvs, as opposed to allowing data to only be either totally private or universally public.

5. The PML must allow for both *dynamic inclusion and exclusion* of common sub-processes, as well as *independent evolution* of private sub-processes. The former is particularly important when independent pre-existing processes decide to collaborate, perhaps only temporarily, while

the latter is important for preserving the autonomy of local processes. The independent operation requirement further implies that the Treaty mechanism should minimize the inter-site dependencies that are required to maintain a consistent Treaty. This point is addressed in Section 2.2.2.4.

In the rest of this section we address requirements 1, 4 and 5. Requirements 2 and 3, which are more database oriented, are covered elsewhere [21].

2.2.2.2 Alternative Approaches

In considering the possible alternatives to expressing common sub-processes within otherwise private and encapsulated processes, we can draw an analogy between our problem and similar problems in the domain of distributed programming languages and systems, and investigate alternatives there:

1. Process interface specified within the PML — This approach includes programming language abstraction mechanisms in which all control and data of a unit are by default private (or hidden) unless specified explicitly as public in the unit's interface. For example, the *body/specification* distinction in **Ada** could be used to expose only the common sub-processes (or sub-*tasks* in Ada terminology) in the specification and hide the private sub-process in the body. Another example is the *export-import* mechanism in **Modula-2**, in which a subset of the activities (functions) could be exported by one process (*module* in Modula-2 terminology) and imported by another, while the rest of the local process (module) is by default hidden.

   The main problem with applying the above approach to our case is that it provides the wrong abstraction. Its prime motivation is to distinguish between a unit's external (public) interface and its internal (private) implementation, promoting modularity, encapsulation, and reuse. While this might be the case in process interoperability, more often the distinction is along the lines of shared versus private sub-processes, regardless of whether the private process is an "implementation" of the shared process. Another problem with this approach is that it is language-based, and thus static in nature, conflicting with the dynamic inclusion and evolution requirement stated earlier. That is, the interface specifications cannot be changed while the program is executing, and all the bindings among the different modules are made at "compile" time.

2. Process interconnection language, separate from a specific PML — This is analogous to dynamic module interconnection languages, in which a separate notation is used to denote how modules are inter-connected. For example, the Darwin [155] configuration language[1] (the successor to Conic [156]) enables (operating system) processes to interconnect independently of the specific language in which they are written, by means of typed *ports* through which data is exchanged between the processes. Ports are protected and made accessible through an import-export mechanism (the actual notation in Darwin is `require` and `provide`).

   The abstraction here is closer to our needs, and it can also be made dynamic, as is the case with Darwin. That is, the nature and kinds of bindings between the processes can be changed dynamically. However, since this is still essentially a language-based approach, dynamic changes impose a problem in terms of comprehensibility: either the changes do not correspond to the original source definitions, which is an obvious problem, or the interconnection is not explicitly declared, defeating in some sense the purpose of using a language-based approach to begin with. The latter approach is taken in Darwin, where the references to the services (or control constructs) are passed in messages, allowing to change their behavior, but as the authors point out, this feature is not recommended for long-term or semi-permanent bindings.

3. Other distributed programming languages — This community produced numerous languages that support some form of dynamic program configuration among relatively independent (operating system) processes. One representative is Hermes [217], another port-based language

[1]Not to be confused with the Darwin environment mentioned earlier.

in which new ports can be added to an executing (operating system) process and existing *port connections* can also be changed, by statements executed from within the existing Hermes code. New processes can also be added using the *create of* statement, but only from within an existing process. Thus, it is not possible to add new facilities that were not anticipated in the original program.

Our Treaty abstraction for defining process-interoperability is different than any of the above alternatives and is geared towards satisfying the requirements. It is defined pairwise between each two SubEnvs that intend to collaborate, reflecting the peer-peer nature of interoperability; it is defined inside each of the participating SubEnvs, to address decentralization; and it allows unilateral cancelation coupled with dynamic verification, to address autonomy. Finally, in contrast to the language-based approaches, we advocate a ***system-based*** approach, i.e., we extend the available PSEE's execution engine with "system calls" that support the definition of the interoperability model. As such, this approach does not require the invention of a whole new process-interoperability modeling language, nor does it make any assumptions about a particular PML, making it generically applicable. (We will return to the issue of language- vs. system-based approach in Section 2.5.2.6.)

2.2.2.3 The Treaty

In the following discussion, the following notation is used:

- E_i denotes an instantiated environment.
- A_i is used to denote a set of process steps that form a common sub-process. Note that in terms of the definition of an environment, A_i is a subset of P (process), i.e., it does not necessarily contain a subset of T (tools), D (data), U (users), but it does imply a subset of S (schema) through the types of the formal parameters to the activities in A_i. Furthermore, A_i may consist of a set of unrelated steps, all of which are part of the common process, or they can be interrelated, for example representing a single common task.
- $A_i(E_j)$ denotes sub-process A_i of environment E_j, i.e., a fragment of E_j's process model.

We define the following operations:

1. $export(A_1(E_1), E_2)$ — Export A_1 from E_1 to E_2, enabling E_2 to *import* A_1. This operation executes locally at E_1.
2. $import(A_1(E_1), E_2)$ — Get A_1 from E_1, and integrate it with E_2's process. This operation executes at E_2 and involves also E_1. A pre-requisite to this operation is that $A_1(E_1)$ was previously exported in E_1. The successful outcome of this operation generates $A_1(E_2)$, a local replicated version of A_1, fully integrated with the rest of E_2's process. The exact meaning of "full integration" is intentionally left out here, since it is PML-specific. Intuitively, the idea is that the newly imported sub-process gets interconnected with the local process and becomes an integral part of that process (Section 2.3.2.3 shows a concrete implementation of *import*). Note that the name of A_1 must be distinct from any other pre-existing or new activity in E_2 so that it can be uniquely identified at runtime.

These operations form the mechanism to implement common activities. However, as mentioned earlier, a separate concern is to determine execution privileges on the common activities, such as which SubEnv is entitled to execute a multi-site activity on remote data. In some cases, invocation of specific activities cannot be made from some of the SubEnvs, for example, due to tool invocation restrictions (e.g., licenses, platforms, location of tool experts, etc.).

It appears at first that such "execution privileges" semantics could be permanently attached to the *export* and *import* operations in some fashion, e.g., to associate a request to execute on remote data with the *export* operation. However, early experiments with our implementation revealed that these are indeed orthogonal concerns that should be distinguished. Thus, we separate the issue of how to provide common multi-site activities from the concern of how to restrict or control their application.

We define the following two execution privileges directives, each of which could be used in conjunction with either of the above operations:

1. *request*(A_1, E_1, E_2) — E_1 specifies an intent to use A_1 on data from E_2. Note that A_1 can be either exported by E_1 or imported from some other SubEnv.

2. *accept*(A_1, E_1, E_2) — E_2 allows A_1 to be used by E_1 on data from E_2. Once again, A_1 could be originally defined at E_1 (or at some third site from which E_1 imported it), in which case it was later imported by E_2, or it could be exported by E_2 and imported by E_1.

To summarize, the four combinations and their intuitive meanings are:

1. *export_request*$(A_1(E_1), E_2)$ — I (E_1) want to use my A_1 on your (E_2) data.
2. *import_accept*$(A_1(E_1), E_2)$ — I (E_2) allow you (E_1) to use your A_1 on my data.
3. *export_accept*$(A_1(E_1), E_2)$ — I (E_1) allow you (E_2) to use my A_1 on my data.
4. *import_request*$(A_1(E_1), E_2)$ — I (E_2) want to use your A_1 on your (E_1) data.

A (simple) **Treaty** (denoted as T) is a binary relationship between two sites, defined as either one of these two possibilities:

$$T_{A_1}(E_1, E_2) = \mathit{export\_request}(A_1(E_1), E_2); \mathit{import\_accept}(A_1(E_1), E_2) \tag{2.1}$$

$$T_{A_1}(E_1, E_2) = \mathit{export\_accept}(A_1(E_2), E_1); \mathit{import\_request}(A_1(E_2), E_1) \tag{2.2}$$

In words, this Treaty allows users operating at E_1 to execute activities defined in A_1 on data from E_2. We shall refer to this Treaty as "a Treaty *from* E_1 *to* E_2 *on* A_1". Both definitions lead to the same outcome, the difference being the origin of A_1: in expression (2.1) A_1 is initially defined in E_1 and is exported to E_2, which imports it; whereas in expression (2.2) A_1 is initially defined in E_2 and exported to E_1, which imports it.

Thus, a Treaty between two SubEnvs consists of one requester and one acceptor, as well as one exporter and one importer. The *export-import* pair of operations establishes a common step (containing multi-site activities), and the *request-accept* pair defines which site is eligible to invoke activities from the common step (the requester) and which one allows access to its data (the acceptor). The gist of the Treaty is that it requires both sides to actively participate in the agreement that determines their inter-process interactions. In particular, a *request* on an activity without a corresponding *accept* on the same activity has no effect on either SubEnv (regardless of whether the activity is properly imported-exported). As for the order of the operations in a Treaty, the main reason for them not being commutative is to protect the privacy of the exporting process. This means that any implementation of *import* should restrict its visibility only to activities which have been already exported to the relevant SubEnv by another SubEnv.

It is important to understand that the Treaty relationship is not symmetric. For example, the Treaty above does *not* imply that E_2 can run activities from A_1 on E_1, i.e., it is only unidirectional. This property of Treaties addresses the concerns raised earlier regarding execution privileges. Furthermore, the Treaty is not transitive, and each Treaty between two sites must be formed explicitly. (Treaties can be considered reflexive, though, if self-export and self-import are defined as "no-ops".)

The extension of a Treaty to multiple sites is defined as:

$$T_{A_1}(E_1, (E_2 \ldots E_n)) = \bigcup_{i=2}^{n} T_{A_1}(E_1, E_i) \tag{2.3}$$

In words, it is the union of all pairwise (simple) Treaties with E_1 as the source SubEnv. This multi-site Treaty allows users operating in E_1 to run activities defined in A_1 on remote data from some or all of E_i, $i > 1$.

To enable symmetric Treaties, we define a (binary) *Full Treaty* (denoted FT) as:

$$FT_{A_1}(E_1, E_2) = T_{A_1}(E_1, E_2); T_{A_1}(E_2, E_1) \tag{2.4}$$

and similarly, a multi-site full Treaty is defined as:

$$FT_{A_1}(E_1, E_2 \ldots E_n) = \bigcup_{i<j} FT_{A_1}(E_i, E_j) \tag{2.5}$$

This consists of the union of all unordered pairs of binary full Treaties (or all ordered pairs of regular Treaties). While symmetric, full Treaties are still not transitive, to protect the privacy of sites as in simple Treaties.

A Full Treaty allows any participating SubEnv to invoke a multi-site activity on data from any other SubEnv in the Treaty. Note that when multiple sites are involved, there are many combinations of possible Treaties between the sites on the same set of activities, not only simple or full. For example, the Treaties:

$$T_A(E_1, (E_2, E_3)) \tag{2.6}$$
$$T_A(E_2, (E_1, E_3)) \tag{2.7}$$

allow either E_1 or E_2, but not E_3, to invoke multi-site activities from A on data from some or all of the three sites.

This model provides maximum flexibility in expressing interprocess collaboration, and each participant in a Treaty must explicitly "sign" it by invoking the proper operation that reflects its role in the Treaty.

In order to withdraw from Treaties, the following operations are defined:

1. *unexport*$(A_1(E_1), E_2)$ — This operation executes in E_1. It removes A_1 from further being available to E_2 and invalidates possible previous Treaties. In addition, it revokes any privileges which were associated with the *export* (see below).

2. *unimport*$(A_1(E_1), E_2)$ — This operation executes in E_2, effectively removing A_1 from E_2's process. Like *unexport*, it invalidates any previous Treaties and privileges which were attached to the *import*.

3. *cancel*(A_1, E_1, E_2) — has the opposite effect of *request*, i.e., it disallows further use of A_1 at E_1 on E_2. It is issued at the requester end of a Treaty.

4. *deny*(A_1, E_1, E_2) — The opposite of *accept*, it disallows E_1 to further access E_2's data through A_1. It is issued at the acceptor end of the Treaty.

Since *export* and *import* are the mechanism for establishing shared common sub-processes, when *unexport* (*unimport*) is executed on a previously exported (imported) activity, the corresponding execution privileges property (either *request* or *accept*) is also revoked (by *cancel* or *deny*). The opposite is not true, though. A *cancel/deny* does not imply *unexport* or *unimport*. For example, a requester activity could be transformed to an acceptor activity by issuing a *cancel* followed by *accept*, regardless of whether it is an exported or imported activity.

2.2.2.4 Local Evolution and Dynamic Treaty Verification

In Section 2.2.2.1 we identified the need to be able to perform process evolutions and Treaty-leaving operations locally with minimum interaction with remote SubEnvs, while still being able to dynamically check the validity of Treaties. We begin with a definition of a valid (or consistent) Treaty, analyze all possible ways in which it can be invalidated, and discuss our dynamic Treaty-verification algorithm.

A (simple) Treaty from E_1 to E_2 on A_1 is said to be valid if and only if all three conditions below hold:

1. Either:
   (a) A_1 is marked at E_1 as exported to E_2, and is marked at E_2 as imported from E_1.
   (b) A_1 is marked at E_1 as imported from E_2, and is marked at E_2 as exported to E_1.
2. A_1 is marked at E_1 as a requester of E_2, and is marked at E_2 as an acceptor from E_1.
3. A_1 is identically defined in both SubEnvs. Since there is no shared space in which Treaties are stored, there must be a way to guarantee that original Treaties have not been altered by the time they are invoked on remote data. We refer to this condition as the "common sub-process invariant".

The first condition is invalidated whenever *unexport* at the exporting site, or *unimport* at the importing site, is issued. When an activity is issued, *unexport* can be easily detected locally at the invoking site — the invocation is rejected if the issued task is not (anymore) exported. *unimport* is also easily detectable since when the task is requested on the remote site, if it is part of a sub-process which has been unimported (and thus removed from the process' set of tasks) the requested activity will simply not be found.

As for the second condition, both *request* and *accept* privileges have to be checked for their validity. E_1 can lose its *request* privileges on A_1 if the equivalent of $cancel(A_1, E_1, E_2)$ was issued. This can occur in one of two ways, depending on the method by which the *request* privileges were originally obtained: (1) If through *export-request*, then an *unexport-request* on A_1 from E_1 to E_2 revokes *request* privileges. This can be validated at E_1 locally when the multi-site activity is invoked, at the same time that the *export* privileges are checked. (2) If through *import-request*, then an *unimport* on A_1 at E_1 invalidates condition 2. Thus, validity checking is similar to that for condition 1.

E_2 can revoke *accept* privileges from E_1 on S_1 whenever the equivalent of $deny(A_1, E_1, E_2)$ occurs at E_2. This can also occur in one of two ways, depending on the original commands issued to set up the privileges. (1) In case of *export-accept*, an *unexport-accept* command revokes the *accept* privileges. To verify this case, E_2 must explicitly check for proper *accept* privileges every time an activity in A_1 is issued from E_1 on data from E_2; (2) In case of *import-accept*, an *unimport* at E_2 invalidates the *accept* privileges. Again, in case of normal *unimport*, there is nothing to check, the activity will simply not be found.

The third condition, requiring identical copies of the Treaty sub-processes at the participant SubEnvs, can become unsatisfied as a result of various (local) process evolutions, and is more complicated to check for. The main problem occurs when a Treaty sub-process is modified at the exporting ("source") SubEnv. Regardless of the process privileges attached to the exported task, such evolution violates the common sub-process invariant.

One method to address this (which was implemented in Oz) is based on *evolution timestamps.* The idea is for the local SubEnv to assign a "timestamp" each time a process is compiled and loaded locally. When a sub-process is imported, its timestamp is also shipped and stored at the importing SubEnv. At run-time, whenever a multi-site activity is invoked for execution, the timestamp at the requesting SubEnv is compared to the one stored at the accepting SubEnv. If there is a mismatch, it means that local evolution took place at the exporting SubEnv, implying invalidation of the Treaty, and the execution is rejected. Re-activation of the Treaty can be made by either re-importing

```
{\it verify-treaty} (TaskId, SrcSubEnv, DstSubEnv):
/* Executes at DstSubEnv */
/* condition 1 */
\ {\bf if} ( find task with the given TaskId )
\ {\bf then}
/* condition 2 */
\ \ {\bf if} ( DstSubEnv $accept$s TaskId from DstSubEnv)
\ \ {\bf then}
/* condition 3 */
\ \ \ {\bf if} (Tasks's remote timestamp $=$ Tasks's local timestamp)
\ \ \ {\bf then}
\ \ \ \ Treaty is valid, allow execution
\ \ \ {\bf else}
\ \ \ \ Treaty is invalid, reject execution
\ \ \ \ Reason: local evolution at the exporting SubEnv
\ \ \ \ Reactivation: re-import (or reload) at the
\ \ \ \ importing SubEnv with proper privileges
\ \ \ {\bf end if}
\ \ {\bf else}
\ \ \ There is no Treaty on that Task, reject execution
\ \ \ Reason: an equivalent of $cancel$ occurred
\ \ \ Reactivation: DstSubEnv needs to $accept$ the Task
\ \ {\bf end if}
\ {\bf else}
\ \ Requested task does not exist in local SubEnv, cannot execute
\ \ (Re)activation: DstSubEnv needs to (re)import the task
\ {\bf end if}
```

Figure 2.3: Dynamic Treaty Verification Algorithm

explicitly the (possibly modified) sub-process, or by reloading the process (perhaps automatically), which also fetches the up-to-date versions of all imported strategies from the exporting SubEnv(s). This dynamic approach toTreaty verification eliminates the need to notify all related SubEnvs when a local process change occurs (some of them might not even be active at that time), and transfers the responsibility of upgrading the imported rules to each individual SubEnv when it actually needs to use them. This "lazy update" approach fits well with the general decentralized philosophy. Figure 2.3 summarizes this section by presenting the dynamic Treaty verification algorithm that is executed in the acceptor SubEnv prior to invocation of each multi-site activity.

2.2.2.5 Treaty Summary

Treaties are the abstraction mechanism used for the definition of process interoperability. The only way by which a SubEnv can collaborate with other SubEnvs is through these pre-defined arrangements that determine how to collaborate, and on what artifacts. Consequently, the degree of collaboration (vs. autonomy) between each pair of SubEnvs is determined by the "size" of their common sub-process. This can range from total isolation (no common sub-process is defined) – where the SubEnvs have no means to access each other's data but are entirely autonomous – to total collaboration (the entire process is common) – where the SubEnvs lose any autonomy and logically share the same process and data and are perhaps only physically distributed.

By splitting a Treaty into two independent operations and the Full Treaty into four operations (as opposed to bundling them to one global operation) we ensure that both ends agree on the Treaty and join it on their own terms. Not requiring synchronous execution of export and import enables Treaties to be formed incrementally and when each party wants to join them. In fact, of all the primitive operations, import is the only operation that requires both sides to be simultaneously active. This independent multi-step protocol also enables SubEnvs to retract from, and join to, a Treaty, independently and dynamically.

Finally, although Treaties are defined pairwise, multi-site Treaties involving an arbitrary number of sites can be formed. It might appear that our approach suffers from being too low-level in that it makes it somewhat complicated to define multi-site Treaties by requiring to form pairwise Treaties. However, this formalism ensures maximum process autonomy. Further, a particular implementa- tion might use "macros" or "scripts" that perform all the necessary operations automatically to form Treaties between "friendly" sites in cases that privacy can be compromised for simplicity and convenience. Alternatively, an implementation may decide to bundle some of the operations into a single built-in command. For example, it could set defaults for combining export and import with request and accept but allow the expert process administrator to modify them. Finally, the PSEE can make provisions for enabling a user to be an administrator on multiple SubEnvs, so that in environments that allow multi-site administrators (e.g., when the interoperability isbetween tightly-coupled SubEnvs), it is possible to bundle the Treaty as one operation, without violating autonomy. Several of these alternatives were in fact implemented in Oz (see Section 2.3).

2.2.3 Multi-Process Execution: the Summit

The Treaty mechanism establishes common sub-processes between sites, and de_nes execution priv- ileges over the common multi-site activities. However, it does not impose a particular approach on how toexecute these shared processes. This is the role of Summits.

2.2.3.1 Alternatives, Design Choices, and Justifications

At first glance, there are two ways in which a multi-site task can be executed: (1) one SubEnv (call it the coordinating SubEnv) copies remote data into its own space and executes locally, or (2) the task leaves the data where it is, and requests that its activities be executed by the remote SubEnvs. This is similar to the two main approaches to distributed program execution: fetch the data and execute locally, or send a request for remote function execution. There are obvious tradeoffs between the two approaches, and the superiority of one over the other largely depends on the nature of the program and the volume of the data involved.

However, since a multi-site task inherently involves more then one process, neither of these approaches is always feasible or desirable: (1) Process autonomy restricts application of the data fetching approach, since some of the remote data might not be accessible to the executing process, and even if it is, the prerequisites and consequences determined by the coordinating process might not maintain consistency with respect to the remote process(es). (2) The function sending approach does not address activities that manipulate data from multiple (local and remote) processes, but instead assumes that an activity's arguments all reside in the same SubEnv. In addition, as mentioned earlier, tools invoked by an activity may not be available at a remote SubEnv (in fact such a scenario might be the initial motivation for running the activity in the originating site), and even copying the tools might not work if the SubEnvs operate on heterogeneous platforms or if there are licensing restrictions.

We devised a third hybrid approach, which combines the two approaches mentioned above in a manner that ameliorates their limitations. Multi-site activities that are defined in a common-sub-process are executed at the coordinating SubEnv by fetching to it all remote data, while local activities emanating in each local process from the common-sub-process are executed locally at each site with local data.

2.2.3.2 The Summit

Following the "international alliance" metaphor mentioned earlier, our decentralized execution model can be described as a "summit meeting". Before the meeting (multi-site activity), each party (process) handles local constraints (prerequisites) that are necessary for the meeting to take place; then the meeting is held at one location (SubEnv), where the various parties send representatives (data) to collaborate; once the meeting is over and agreements were made (results of the activities), all parties return home (to their SubEnvs) and carry out the implications (consequences) of the meeting locally. Summits can lead to subsequent Summits, each involving a subset of the parties, possibly with different representatives (data arguments). It is important to note that each of the two metaphors, namely Treaty and Summit, are independent from each other in our model. That is, whereas in the international community Summits (may) lead to Treaties, in our model Treaties actually enable Summits.

Process interoperability takes place when an activity is invoked (either manually by an end-user or automatically by the process engine) on data from one or more remote SubEnvs. (The case of only local data from the same SubEnv does not lead to inter-process collaboration, and is handled however it would normally be by the underlying single-process PSEE.) We call the process from which the multi-site activity is invoked the *coordinating process*. The Summit protocol consists of the following phases:

1. *Summit Initialization and Treaty Verification* — The coordinating process in which the Summit request was issued establishes a task context (necessary to support interleaved execution of multiple activities) and allocates the necessary resources needed for the Summit. It then binds the actual parameter objects (at least one of which is remote, or otherwise this would not be considered a Summit) to the formal parameters of the activity. Initialization is followed by executing the Treaty-verification algorithm shown in figure 2.3.

2. *Pre-Summit* — The involved processes (i.e., those that own some of the data requested by the multi-site activity) are notified, and all of them (including the coordinating process) perform

simultaneously and asynchronously pre-Summit process actions, *each according to its local process, with its local data and tools, in the local SubEnv.* Pre-Summit actions include: (1) Verification that prerequisites imposed by the process step enclosing the activity are satisfied locally; this may be regarded as "internal" constraints. (2) Verification that the activity can be executed with respect to the overall task workflow; this may be regarded as "external" constraints (see [127] for more on this distinction). (3) Active invocation of related activities, e.g., to satisfy (1) and (2). And (4) Deriving and binding data arguments that are required by the activity but were not specified as parameters. Note that Pre-Summit requires that all involved SubEnvs identify the same requested activity, in order to know what to verify/satisfy. This is guaranteed through the *import* mechanism of the Treaty.

One optimization that can be made in some cases (depending on the PML as well on the specific activity) is for the coordinating process to determine locally whether or not launching a remote pre-Summit is necessary for each participating SubEnv, in which case no "fan-out" to the local sites is required. In general, however, the local SubEnvs need to be able to decide for themselves whether or not they need to undertake any work. The main point is the locality of the execution, which is determined solely by each SubEnv on its local data, without "global" intervention.

3. *Summit* — If pre-Summit is successful in all involved processes, the requested activity is invoked in the coordinating process, with all the necessary local and remote data arguments. The activity is executed synchronously, and it may or may not execute at one location depending on the kind of tools associated with the activity. For example, it may launch a cooperating set of tools, on one or more sites, involving one or several users.

4. *Post-Summit* — When the Summit completes, all involved SubEnvs are notified, and all of them (including the coordinating SubEnv) perform simultaneously and asynchronously post-Summit process actions, again *each according to its local process, with its local data and tools, in the local SubEnv.* Post-Summit actions include: (1) Assertions on the process and product data that reflect the fact that the various activities were executed (depending on the PSEE, it may not always be possible to directly modify such data within the activities themselves); (2) Binding and assignment of data affected by the activities that were not supplied as arguments; (3) Verifying that consequences imposed by the steps in the Summit can be fulfilled (this is not always a logical implication of the pre-Summit verification); and (4) Triggering execution of further activities, e.g., as part of (3).

5. *Summit Completion* — When post-Summit completes in all local sites (including the coordinating SubEnv operating in "local" mode) the coordinating SubEnv checks whether further Summits are pending (see below). If any Summit activity is pending, the algorithm returns to step 1. If no Summits are pending, the Summit is completed by releasing all resources associated with the Summit.

Thus, both pre- and post-Summit phases occur asynchronously in each SubEnv *only* according to its local process, while execution of the Summit phase occurs synchronously and involves collaboration among the participating SubEnvs. This design minimizes the interference between the processes (and hence maximizes autonomy) while still allowing them to carry out the desired common activities as agreed upon in the Treaty.

A composite Summit (i.e., consisting of multiple Summit activities) can be viewed as alternating between "local" mode — whereby each participating site (including the coordinating site) performs local operations asynchronously — and "global" mode in which the coordinating process synchronously carries out operations involving data from multiple sites, with the approach intended to minimize the "global" mode and maximize the "local" mode.

2.2.3.3 Example

The following example illustrates the execution of Summits. Assume there are three development teams working in separate sub-environments **SE1**, **SE2**, and **SE3**, who are responsible for three disjoint components of a system **S**, labeled **S1**, **S2**, and **S3**. The teams operate at different sites and reside in different geographical areas. They each work on their own artifacts (e.g., files, documentation) using their private tool set and their own processes. Each component can be coded and unit-tested independently, and the components are interconnected through published, well-defined, interfaces. Suppose **S2**'s interface has to be modified in order to enhance some of its functionality, thereby requiring the other components to change. The following steps should be taken: (1) the proposed change has to be reviewed and approved by all SubEnvs; (2) the interface of **S2** is actually modified; (3) The affected components are modified to correspond to the new interface; (4) a local test of each component is performed; and (5) an integration test with all revised components is performed. For simplicity, only the "successful" path, i.e., assuming that all the steps were carried out successfully, is described. While the global modification and integration test must be performed synchronously (with respect to all sites) and at one site, the review, local modification, and local test activities can be performed asynchronously in the local sites, and they can differ at different sites. For example, one site might employ "white box" local testing, while another site might use "black box" testing. Moreover, even identical operations might trigger different related operations when issued at different sites.

Figure 2.4 illustrates the enactment of this example as a (composite) Summit. The `change` activity is initiated by the coordinating SubEnv **SE2**. Pre-Summit takes place in a decentralized manner, where each SubEnv performs the `Review` activity locally according to its own process. For example, **SE3** requires an additional `analysis` step before the review and both **SE1** and **SE2** require a check-out phase using different configuration managers (RCS and SCCS, respectively). Once reviewed by all sites, the Summit activity `approve` is executed, determining whether to approve or disapprove the change based on the local reviews. If the approval step succeeds, the `modify` activity is executed, where the objects are modified. When finished, post-Summit begins, again in a decentralized manner. All SubEnvs are engaged in a unit-test step, but each one does it according to its own process. For example, **SE3** employs a manual-test procedure (e.g., for testing the user interface) which involves human users that actually perform the tests (devising the input sequences for the test suites can be also done manually or automatically for either manual or automatic testing), whereas the other SubEnvs perform automatic testing, but **SE2** has an additional code-inspection step. Completion of the local testing leads to `integration-test`, another Summit activity in this composite Summit.

It is important to understand that Figure 2.4 depicts one particular execution trace of the process, not the whole process. For example, a different execution would occur if the `review` activity failed at **SE1** (i.e., the reviewer did not accept the proposed change). In this case, a revision phase would be followed, after which a second review would be scheduled, and so forth, until the review succeeds.

2.3 Realization of the Interoperability Model in Oz

The generic model, as a high-level abstraction, leaves many aspects undefined and unresolved, both technical and conceptual. We address some of these issues here by discussing the realization of Treaties and Summits. The architectural aspects of Oz, including site interconnectivity, configuration, transactions, database and cache management, are beyond the scope of this report and can be found elsewhere [21, 26].

2.3.1 Oz Overview

Oz is a multi-process PSEE (as defined in Section 2.2.1.3), and it supports definition and execution of autonomous multiple SubEnvs following the Treaty and Summit models. When not interoperating with other SubEnvs, the functionality of a local SubEnv resembles that of Marvel [31], the predecessor

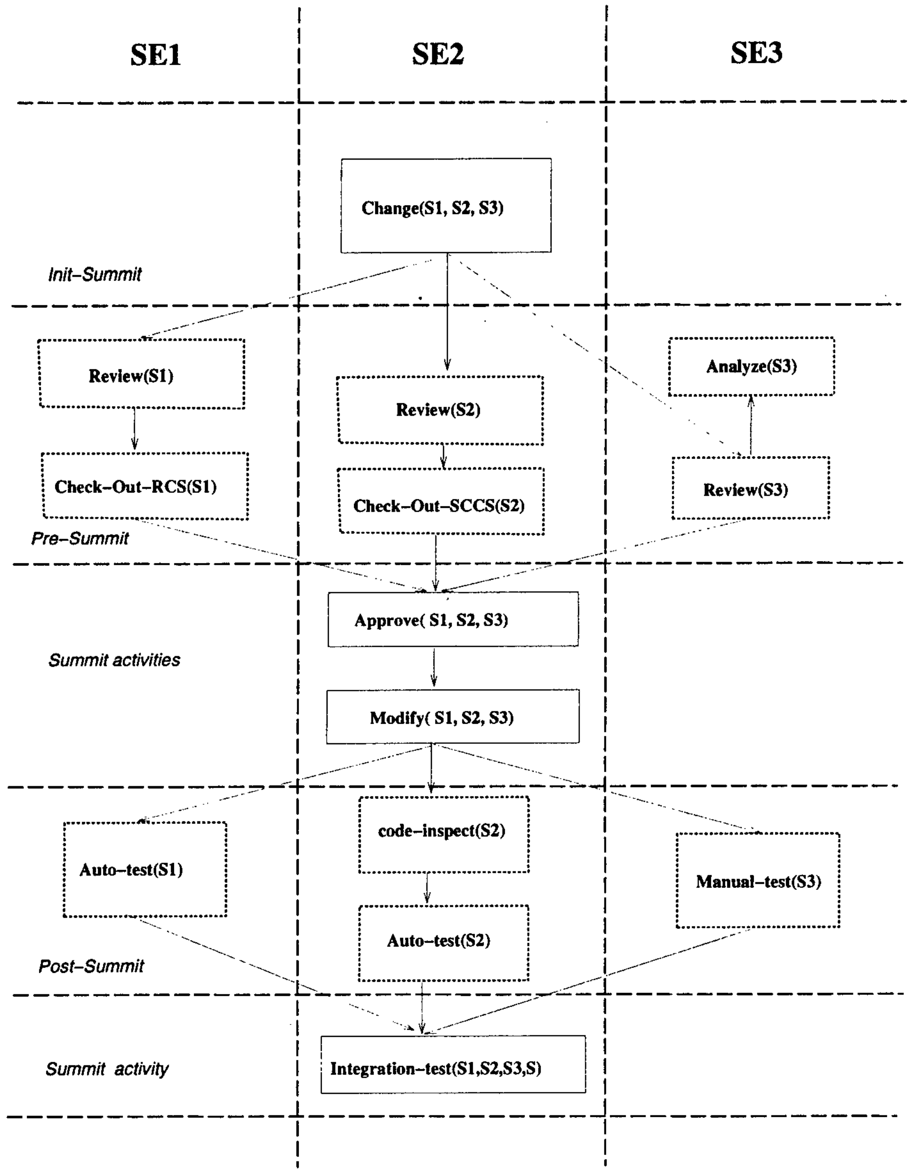


Figure 2.4: An Example Summit

to Oz. As in Marvel, each *local* (sub)environment in Oz is tailored by a local administrator who provides the data model, process model, tool envelopes and coordination model for its team. These definitions are translated into an internal format and then loaded into the environment. The process modeling language of Oz is based on the Marvel Strategy Language (MSL) [142]. Most importantly, Oz extends the user-driven, rule-based paradigm to multi-process environments. Specifically, as far as local processes are concerned, Oz processes are defined in terms of rules which correspond to the notion of process-steps in the generic context hierarchy.

A rule consists of: a signature, i.e., names and types of formal parameters; a binding section, where additional objects (termed derived parameters) are bound to the rule as a result of querying the database; a pre-condition consisting of a (composite) predicate over the arguments; an activity which interfaces to external tools and data; and a set of mutually-exclusive effects. A rule can be fired either directly by a user (via a client, see below), or indirectly, as a result of rule chaining. When a rule is fired, its condition is evaluated. If the evaluation fails (i.e., the predicate evaluates to false), the rule processor attempts to automatically satisfy the rule by backward chaining to other rules whom effect may satisfy the failed condition, recursively. If the condition is (or has become) satisfied, the activity of the rule is spawned and executed on behalf of the user who invoked the rule (or, in case of a chained rule, the user who issued the rule that chained to this rule). Upon completion, the activity returns a return code that determines which effect of the rule to assert. The rule processor then attempts to forward chain to rules whom condition have become satisfied as a result of the assertion, recursively. Thus, process steps are implicitly interrelated by logical matchings between effects and conditions of rules. In order to enable finer control over the degree of chaining, several chaining directives can be applied on rule predicates. For example, a `no_forward` directive on a rule's effect disables any forward chaining from that rule.

Oz has a two-level architecture: within a SubEnv, it has a client-server architecture with multiple clients communicating with a single centralized process-server. Across SubEnvs, Oz has a multi-server "share-nothing" architecture, as advocated in the formal model. This means that the processes, schemas, and instantiated objectbases are kept separately and disjointly in each SubEnv, and that there is no global repository or "shared memory" of any sort.

Human interaction with the environment is provided through a *client* that is connected primarily to its local *server*. Using the client's connection to its local server, users can operate the local tools (encapsulated in rule activities), on local data objects, under the local process. In addition to the local server, however, Oz users can connect to remote servers. Each remote SubEnv is represented in each local objectbase by a "stub" object that is visible to the client. By issuing the built-in `open-remote` (`close-remote`) command with the appropriate stub object as parameter, a client can open (close) a connection to a remote SubEnv. A remote connection provides limited access to the remote SubEnv. A remote client can browse through remote objectbases and get information about remote objects (subject to access control permissions). However, a client has no access to remote processes (i.e., rules, tools) and manipulation of remote data can be done only by binding remote objects as parameters to Treaty rules.

For example, figure 2.5 shows how the client for user `israel`[2] is connected to the local server of SubEnv `NY`, with a (default) view of the local objectbase[3] (parent-child relationships are depicted with straight lines and links by curved lines). Figure 2.6 shows `israel`'s view after an `open-remote` on site `CT` has been made, making `CT`'s remote objectbase available for browsing by `israel`. `israel`'s client has not connected to SubEnvs `MA` and `NJ`, and they may, or may not, be currently active (i.e., executing). `israel` interacts with the environment by selecting commands from the `rules` menu, which contains all the process-specific user-level commands, and he supplies arguments to the rules by clicking on objects from the objectbase. In particular, if a remote objectbase is open, `israel` can initiate a Summit by selecting remote objects as arguments to Treaty rules. When the (local) server services the request to fire a rule, it checks its own process, and communicates with remote SubEnvs if the rule accesses remote data from their objectbases, and eventually determines whether

[2]The user's name is shown in the upper left corner of the interface window.

[3]For simplicity, only a small objectbase is shown, but in reality Oz can maintain thousands of objects with adequate browsing support.

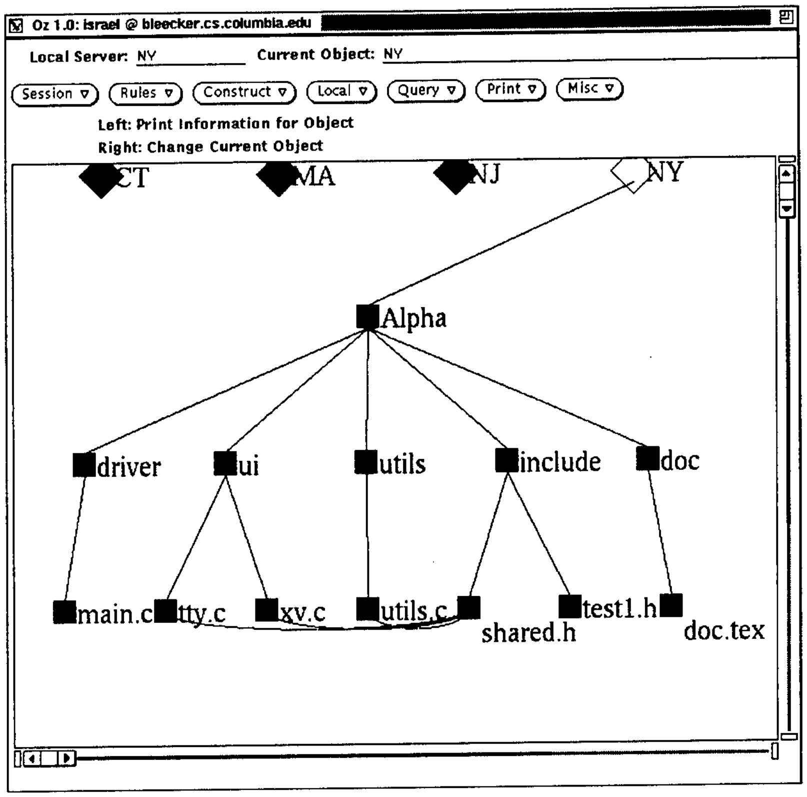


Figure 2.5: An Oz Environment

an activity has to be executed. That activity could be either the one explicitly requested by the user, or another activity related to the requested one through a chained rule. The server then sends a message to the requesting client to execute the activity in its activity-manager component. During a Summit activity, remote objects are temporarily copied to the local SubEnv and passed to the client prior to the activity execution. Note that since a client has no explicit access to remote processes, it cannot invoke "remote Summits", thus all Summits are initiated by local clients.

2.3.2 Treaty in Oz

Treaties in Oz follow the formal Treaty model. The basic unit of commonality in Oz is the *rule*. However, as a "syntactic sugar", the unit that is exported and imported is the *strategy*, a bundling construct for rules, somewhat analogous to a module consisting of functions in modular programming languages.

Oz provides five built-in commands for establishing Treaties: *export*, *import*, *unexport*, *unimport*, and the "non-standard" *treaty* operation. Although there are no separate commands for *request*, *accept*, *deny* and *cancel*, they are specified as parameters to each of the above commands, making it possible to generate all possible combinations that were discussed in the formal model in Section 2.2.2.

2.3.2.1 export

The *export* operation is defined as:

$$export(strategy(SrcSubEnv), DstSubEnv, [privileges])$$

It executes locally at *SrcSubEnv* and merely involves adding an entry with the specified *strategy* and *DstSubEnv* to a persistent local export table. By default, Oz associates *request* privileges

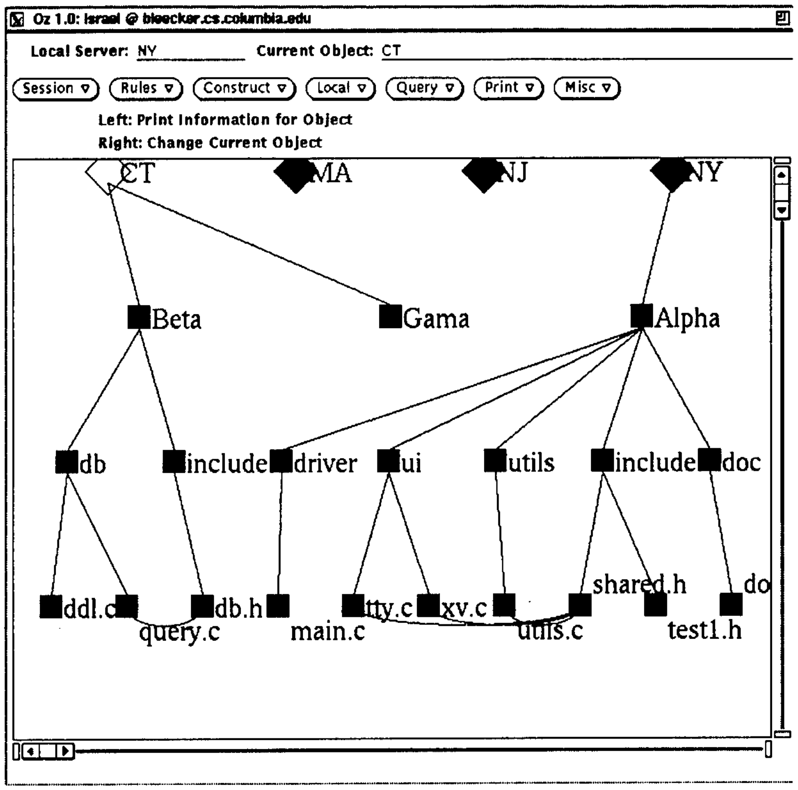


Figure 2.6: Oz Environment with one open remote site

with *export*, i.e., it assumes that in most cases the exporter wants to use the exported strategy on data from *DstSubEnv*. But the administrator can change the default by explicitly selecting *accept* privileges. In addition to *accept* and *request*, Oz provides a third option called *shared*. The semantics of the *shared* option are to export a strategy both as a requester and as an acceptor. The main use of this option is to facilitate convenient generation of full (i.e., bi-directional) Treaties: a *shared export* followed by the proper *shared import* establishes a full Treaty.

2.3.2.2 unexport

The *unexport* operation is defined as:

$$unexport(strategy(SrcSubEnv), DstSubEnv, [privileges])$$

Like *export*, this is a local operation that executes at *SrcSubEnv*. It removes *DstSubEnv* from the list of SubEnvs that are entitled to further import *strategy*. In addition, the execution privileges are undone based on the specified *privileges* argument — when coupled with *accept* the effect is *deny*, coupled with *request* results in *cancel*, and coupled with *shared* revokes both. Note that if, for example, the exported strategy was previously *shared* (i.e., both requested and accepted), then unexporting with *request* (*accept*) retains the *accept* (*request*) privileges intact.

2.3.2.3 import

The import operation is defined as:

$$import(strategy(SrcSubEnv), DstSubEnv, [privileges])$$

import is the main operation in Treaties. We assume the existence of the necessary underlying infrastructure to communicate with the remote SubEnv (This topic is beyond the scope of this report, see [21]). In particular, there must be a connection from *DstSubEnv* to *SrcSubEnv*, since the operation is initiated at *DstSubEnv* but it involves both SubEnvs. The realization of *import* consists of four distinct phases:

1. *Select* — Since remote strategies are not normally visible to SubEnvs, the *import* interface must supply the administrator at *DstSubEnv* with a list of the available strategies at *SrcSubEnv* that were explicitly exported from it to *DstSubEnv*. Further, this information must be generated dynamically, since the list of exported strategies at *SrcSubEnv* can change at any time as a result of issuing local *export* or *unexport* operations.

2. *Copy* — Once the importer at *DstSubEnv* selects the strategy to import, the strategy is copied from *SrcSubEnv* along with additional information needed for runtime validation (e.g., timestamp). The source-code of the strategy is used only during the integration phase, however, and cannot be manipulated by administrators at *DstSubEnv*, to ease dynamic verification of Treaties.

Note that *import* fetches only the rules, without the tools and their envelopes (i.e., the wrapping mechanism used to integrate tools into Oz, see [86]). While this is not a problem with the default *import-accept* option (where the activity is not executed at the importing SubEnv, only its data is accessed by the activity, which executes at another SubEnv), the *import-request* combination implicitly assumes that either the activity and its associated tools already exist at the importing SubEnv, or they can be copied explicitly. If this is not the case (e.g., a tool is bound to a specific machine and cannot be copied), then this combination should not be used.

3. *Integrate* — This is the main step. First, the imported strategy is parsed and checked to be schema-compatible with the local process. Next, the rules in the parsed strategy are integrated with the rule network, by *forward* connecting each new rule to all other rules (both imported and local) whose conditions match the rule's effect, and *backward* connecting it to all rules whose effects matches the rule's condition. At the end of this procedure, the imported strategies are fully integrated with the local process. When executed as part of a Summit, local prerequisites and consequences (in addition to "global" Summit implications) of the imported Summit rules would be automatically enacted.

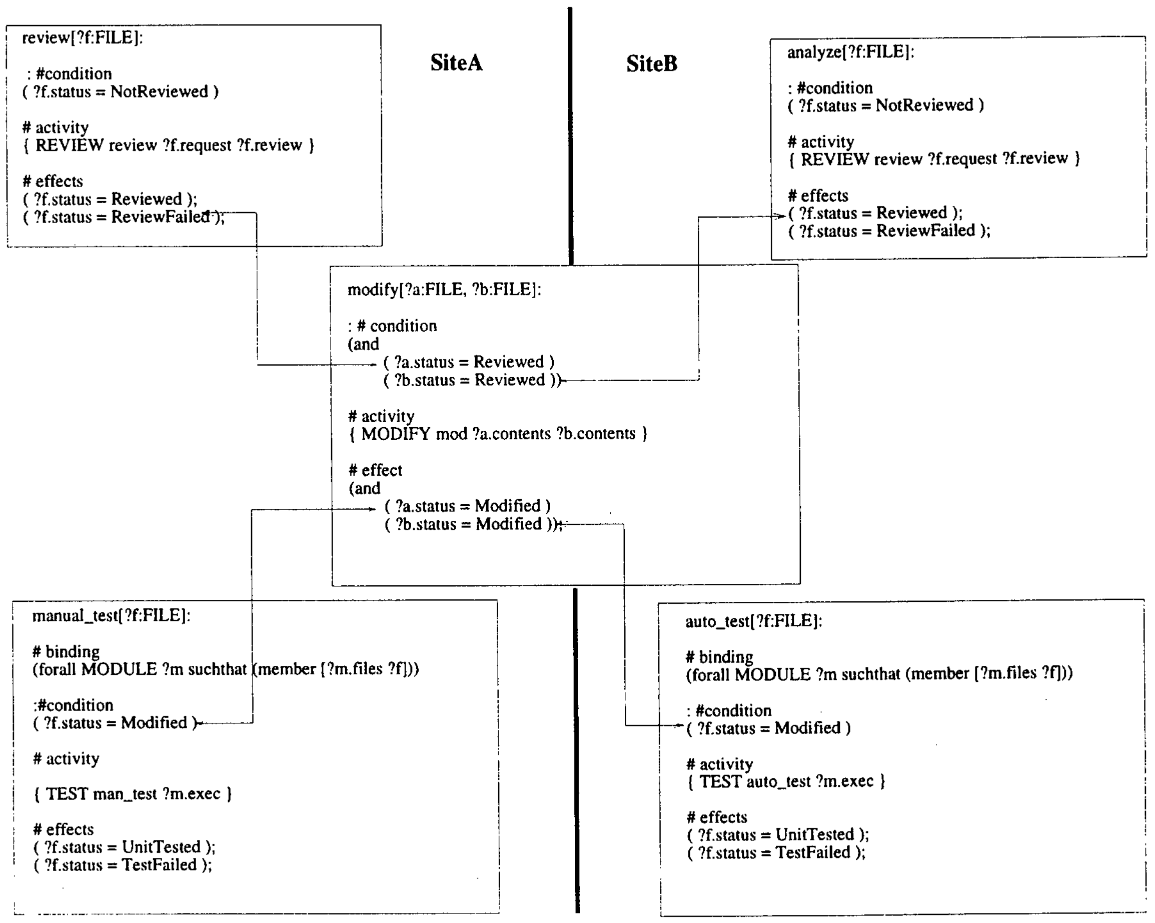
Figure 2.7 illustrates the integration phase. Suppose the `modify` rule is imported by two different sites, `SiteA` and `SiteB`. In `siteA`, `modify` is backward connected to rule `review` through the matching between `modify`'s condition and `review`'s effects, and it is forward connected to rule `manual_test` through the matching between `modify`'s effects and `manual_test`'s condition. In `siteB`, the rule `modify` is backward connected to `analyze` and forward connected to `auto_test`. Thus, `modify` becomes an integral part of both processes, and may trigger, or be triggered by, invocation of related rules during execution. Notice that in general an imported rule may connect to zero, one or more local or other imported rules.

The ease with which process integration can be achieved reveals the strength of the declarative nature of the rule paradigm: process fragments can be incrementally added (or incrementally removed) and *automatically* integrated without user intervention. The context-less rules, as well as the fine granularity of rules as process building blocks, also pay off handsomely.

Due to the coupling of *import/export* with *accept/request* in Oz, it is necessary to make *import* idempotent with respect to the compilation mentioned above, and to allow a SubEnv to export an imported strategy. This is particularly important for multi-site Treaties. For example, suppose site E_1 imports strategy S_2 from site E_2 and site E_3 also imported S_2 from E_2. Now site E_1 wants to grant *accept* privileges to E_3, so it issues an *import-accept* command, but this time compilation of the process model is not necessary so only execution-privileges are modified. When an *import* is requested on an already imported strategy (or alternatively, if it is a local strategy which was exported and is now imported, possibly to form a full Treaty), only the process privileges are updated, and the compilation part is ignored. We refer to such operation as a "faked" *import*.

4. *Acknowledge* — An acknowledgment is sent to *SrcSubEnv*. This acknowledgment is not critical, however, since Treaties are verified at runtime. Its sole purpose is to notify users at *SrcSubEnv* of the new Treaties that are available to them.

There are two more properties that the *import* operation must possess. One is *atomicity*; clearly, the *import* operation has several potential failure points, meaning that it must be accompanied by



```
review[?f:FILE]:

 : #condition
( ?f.status = NotReviewed )

# activity
{ REVIEW review ?f.request ?f.review }

# effects
( ?f.status = Reviewed );
( ?f.status = ReviewFailed );
```

```
analyze[?f:FILE]:

: #condition
( ?f.status = NotReviewed )

# activity
{ REVIEW review ?f.request ?f.review }

# effects
( ?f.status = Reviewed );
( ?f.status = ReviewFailed );
```

```
modify[?a:FILE, ?b:FILE]:

: # condition
(and
      ( ?a.status = Reviewed )
      ( ?b.status = Reviewed ))

# activity
{ MODIFY mod ?a.contents ?b.contents }

# effect
(and
      ( ?a.status = Modified )
      ( ?b.status = Modified ));
```

```
manual_test[?f:FILE]:

# binding
(forall MODULE ?m suchthat (member [?m.files ?f]))

:#condition
( ?f.status = Modified )

# activity

{ TEST man_test ?m.exec }

# effects
( ?f.status = UnitTested );
( ?f.status = TestFailed );
```

```
auto_test[?f:FILE]:

# binding
(forall MODULE ?m suchthat (member [?m.files ?f]))

: #condition
( ?f.status = Modified )

# activity
{ TEST auto_test ?m.exec }

# effects
( ?f.status = UnitTested );
( ?f.status = TestFailed );
```

Figure 2.7: Integration of Imported Rules

a context-sensitive rollback mechanism that preserves the integrity of the server in case of failures. However, since the acknowledgment phase is optional, there is no need to guarantee cross-site atomicity for *import*. The atomicity of the operation has to be preserved only in the importing server. This fits well with the general decentralized requirements.

The second required property is *persistence*. The imported compiled strategy, along with the necessary information used for runtime verification, must be stored permanently with the local process since it outlives an execution of the server, and needs to be reloaded in subsequent evolutions.

2.3.2.4 unimport

The *unimport* operation is defined as:

$$unimport(strategy(SrcSubEnv), DstSubEnv, [privileges])$$

Unlike its *import* counterpart, *unimport* is a local operation. However, unlike *unexport*, it might involve some non-trivial amount of work at the server. The algorithm is as follows: if *strategy* is marked as imported from more than one SubEnvs, or if *strategy* is a local strategy (which was "faked" imported for full Treaty purposes), then *unimport* does not modify the process, and only updates the privileges similar to the way it is done in *unexport*. If, however, *DstSubEnv* is the only site from which *strategy* is marked as imported, then *unimport* removes *strategy* from *SrcSubEnv*'s process. This requires "decremental" recompilation and regeneration of the rule network. Such an *unimport* also revokes all privileges from all remote SubEnvs regardless of the parameters that were specified with the operation, since the strategy is removed from *SrcSubEnv* and cannot be used in any manner there.

As can be seen, not having the four execution-privileges commands (*request*, *accept*, *cancel*, and *deny*) available separately from the four strategy-transfer commands (*export*, *unexport*, *import*, *unimport*) introduces some technical and conceptual difficulties. On the other hand, preliminary experiments showed that easing the procedure of forming Treaties is pragmatically important, and that most of the Treaties can be formed using the default privileges, while more proficient administrators can still select other options in order to get the desired behavior. In any case, this is mainly a user-interface issue; the main point is that the equivalent semantics of the formal model are fully obtainable in Oz.

2.3.2.5 Forming Treaties

Going back to the formal model, a simple binary Treaty between two SubEnvs is formed by an *export* operation at the source SubEnv, followed by a matching *import* operation at the target SubEnv. But these operations do not have to be synchronized, and in particular, the *import* can occur at anytime after the *export*, or never occur at all. From the system's standpoint, Treaties are formed *implicitly*, and perhaps even without explicit intention. That is, Treaties can be inferred automatically, when the right combination of *export* and *import* occurs at the SubEnvs. In some sense, this is a continuation of the context-less rule-based model that fits well with autonomy concerns. In particular, there is no need for a "global administrator" to form Treaties; they are formed by local administrators willing to collaborate in order to form the Treaties, and using the system to formalize their intentions as well as to ensure that they are carried out as agreed.

In cases where SubEnvs are more tightly coupled, however, there might be a need to support (simple and full) Treaties as one operation, to simplify their formation. Indeed, early experience with Oz revealed the need for such an operation in cases where, for example, each SubEnv represented a single-user process as part of a multi-user global environment, in which case a global administrator (and a corresponding global Treaty operation) was essential. Therefore, Oz supports the explicit *Treaty* operation, which bundles *export* and *import*, as explained below.

In order to be eligible for executing a Treaty operation, a user has to have administrator privileges on both SubEnvs. Note, however, that in conformance with the "not-only-local-or-global" principle, the user does not need universal administrator privileges, only on the two sites of a given Treaty.

The treaty operation is defined as:

$$Treaty(strategy(SrcSubEnv), DstSubEnv, [privileges])$$

The semantics of the operation are as follows: ***strategy*** is exported from ***SrcSubEnv*** and subsequently imported by ***DstSubEnv***. Treaty is atomic, meaning that both SubEnvs have to rollback in case of a failure. In addition, ***DstSubEnv*** has to operate in single-user mode (i.e., only one client can be connected to it, although ***SrcSubEnv*** and other SubEnvs might have arbitrary number of active clients). To simplify matters, Treaty is always initiated by the exporter. However, the exporter can be either a requester (default) or an acceptor, implying acceptor or requester privileges on ***DstSubEnv***, respectively. Finally, as mentioned earlier, a ***shared*** privilege implements a full Treaty, i.e., either site can operate the rules in the strategy on the other site's data.

2.3.3 Summit in Oz

Summits are the main means by which multiple SubEnvs actually interoperate, and as such, they encompass all the support that is required to enable execution of multi-process "Treatified" tasks. Thus, whereas Treaties refer to static properties of rules and data (e.g., formal parameters and types), Summits are concerned with dynamic properties of rules under execution, such as the runtime objects that are bound to an executing rule, the chaining context in which they execute, and so forth.

2.3.3.1 Summit Initialization and Treaty Verification

A Summit task is initiated as a result of an explicit request from a user. From the user's point of view, the only difference between invoking a Summit rule and a normal rule is that at least one of the parameter objects specified by the user is remote. The first action taken by the coordinating server is to fetch copies of the remote objects from their origin SubEnvs, and bind them to the parameters of the rule (in addition to the obvious binding of local objects, but as we focus here on inter-site issues we will ignore from now purely local aspects).

The second step involves Treaty verification. The coordinating server checks locally whether the rule could be invoked as a Summit rule, by checking that the rule has ***request*** privileges on the remote participating SubEnvs (i.e., those SubEnvs that have objects bound to parameters of the rule). If this is not the case, the rule cannot be executed in a Summit. But, as explained earlier, this is only a necessary condition, not a sufficient one, because the Treaty might have been invalidated unilaterally by one or more of the participating remote SubEnvs. So, after local verification, the coordinating server requests each participant SubEnv to execute the verification algorithm from figure 2.3.

The reader might wonder why is it necessary to fetch the remote objects before doing Treaty verification. The reason is somewhat pragmatic, and has to do with the rule-overloading mechanism. Oz allows multiple rules with the same name to co-exist, and determines which rule to execute based on the types and number of actual parameters supplied by the client [108]. Thus, when the local server receives a request to execute a rule, it has to find the "closest" rule that matches the types of the parameters, so only after the remote objects (and their type information) are fetched, can the server determine which rule is intended for the Summit.

2.3.3.2 Pre-Summit

The coordinating server evaluates the rule's condition. If the condition is not satisfied, the server fans out to the participating sites and triggers local backward chaining at each site in an attempt to update the objects so that they satisfy the condition. Backward chaining is private, i.e., each process performs this step according to its autonomously defined sub-process.

One important aspect of remote backward (and also forward) chaining involves execution of remote activities. In Oz, both backward and forward chaining can lead to the execution of further activities, since the chained-to rules are regular rules that may contain arbitrary activities. In particular, some of those activities might be interactive, requiring input from a user. This presents both conceptual as well as technical problems that do not come up in local backward chaining: conceptually, the remote server must determine which user's client should execute the remote activities; technically, it should be able to redirect the activity to the specified user's client. The solution in Oz is to direct all activities to the initiating user, by default. An optimization could be to direct only interactive activities to the remote client and execute non-interactive activities with a local "proxy" client (see [226]). To provide a full solution, however, Oz allows remote activities to be delegated to (remote) users by extending its modeling language to specify delegation, and by providing a delegation mechanism that redirects activities. This topic is beyond the scope of this report, see [21].

2.3.3.3 Summit Activity

If the condition of the rule is satisfied, the multi-site activity is fired at the coordinating site. Since typical Oz activities involve (possibly large) files — as opposed to pre- and post- phases which access "light" process state information — multi-site activities require sites which are physically remote to transfer the remote files to the coordinating server.

Another issue regarding multi-site activities is the association with users. In case of a single-user activity, Oz associates the activity with the user whose on behalf the Summit rule was invoked, or to a delegated user if it was specified in the rule. In case of a multi-user groupware activity (e.g., virtual whiteboard), Oz provides mechanisms to define the participants and bind the activity to them at run time, see [29].

2.3.3.4 Post-Summit

The first step in Post-Summit asserts the appropriate (local and remote) effects of the Summit rule, depending on the output from the activity. Since the executed rule is identical at all participating sites (because of the common sub-process invariant), this phase can be carried out in one of two ways: either the coordinating server sends a message to the remote servers to assert the effect of the rule on the objects (which are remote to the coordinating server and local to each remote server), or the coordinating server itself asserts the effects on the replicated objects and sends the updates to the remote servers. The latter approach simplifies rule processing in that the Summit rule executes as a whole at the coordinating server and there is no need to invoke remote rule processors to execute rule "fragments". In addition, the replicated remote objects must be updated in the coordinating server anyways for object cache management. Therefore, Oz employs the latter approach. In order to enable forward local chaining, the coordinating server sends, in addition to the object updates, a pointer to the asserted effect, and each of the remote servers acts as if it had asserted the effects locally to explore forward chaining possibilities.

Following the derivation phase, forward fan-out takes place. Each SubEnv then determines which rules to execute based on its local process, and carries out the chains locally until all possible forward chains have completed. At this point, they return to the coordinating server.

2.3.3.5 Inference of Summit Rules

There are several approaches to modeling and enacting multi-step Summit rules. Technically, the coordinating server must distinguish chains which are part of the local fan-out from those which are "global" Summit rules. One alternative is to add "Summit" directives, similar to chaining directives, that explicitly annotate effect predicates in rules as "Summit" predicates. These annotations could be used to determine which chains are local, and which are global. In fact, the initial implementation in Oz was done that way. However, this alternative both limits the power of the rule inference engine and proves to be unnecessary.

Given that a Summit rule is syntactically a "normal" rule that just happens to have remote objects bound to it, then by extending the mechanism for dynamic binding of parameters [108] to handle binding of both local and remote objects to chained rules, the basic rule-inference mechanism can infer Summit rules — these are simply the rules that happen to have been instantiated with (some) remote objects as parameters. Thus, the inference of Summit rules has been extended to operate in the same manner as local inference is done. However, unlike normal rules, when Summit rules are inferred they are enqueued in a separate Summit queue and are scheduled for execution only after local forward chaining has completed in all sites that are part of that Summit (see below).

The main advantage of this approach is that as a natural extension of the rule processor for handling derivation of Summits, it is no more (and no less) implicit that derivation of rules, and it has the potential for *automatically* inferring multi-step Summits which could not have been formed in the explicit notation unless they were pre-determined. Another advantage is that Summit rules are formed only as needed, whereas the annotation approach would force the administrator to consider Summits even when no remote data is involved. Finally, adding annotations would have added an (apparently unnecessary) burden on process administrators in forming Treaties.

2.3.3.6 Summit Completion

Once local forward chaining completes in all involved SubEnvs, they notify the coordinating server, which in turn checks if there are any rules in the Summit queue. If there are none, it completes the task and releases resources that were allocated for the Summit (e.g., transaction locks, which are beyond the scope of this report, see [21]). If there are pending Summit rules, the coordinating SubEnv reiterates to the Summit initialization phase, except it bypasses the manual parameter binding phase which is (automatically) performed by the extended parameter binding mechanism. Recall that binding must occur before the initiation of forward Summits, because it is the binding phase that actually recognizes which rules are Summit rules.

2.4 Application of the Model to Other PMLs

We now outline how the interoperability model may be applied to two other families of PSEEs categorized by the paradigm underlying their PMLs, namely Petri nets and Grammars (application to imperative process programming such as APPL/A can be found in [21]). These families, together with rules, cover most kinds of PSEE [132]. Since we take the existing PMLs as given, the uninitiated reader should see the cited references for background and justification of each approach to process modeling.

2.4.1 Petri Nets

The Petri net [178] is a formalism for modeling concurrent systems, and it has been widely applied to software process modeling. The application of our decentralized model to Petri net-based PSEEs is influenced primarily by SLANG [10] and FUNSOFT [90], and their corresponding PSEEs, SPADE and MELMAC, respectively. Each of these PMLs is based on extended Petri net formalisms (specifically, SLANG is based on ER nets, and FUNSOFT on predicate/transition nets), but we will use for the most part general Petri net terminology.

Transitions usually represent our notion of activities (note that our activities are different from SLANG's notion of activities, which are more like our notion of a task). The equivalent of a process activity that involves (possibly external) tools is termed in SLANG a black transition, and in FUNSOFT it is called a regular agency.

Places represent the activity's formal parameters, and *Tokens* represent the current state of the process under enactment and the product data used in the activities (i.e., the actual parameters).

A *predicate* can be attached to a transition and must be satisfied prior to firing the transition. The predicates define local constraints on an activity, as opposed to the general control flow expressed by the topology of the net. Both languages support the notion of a predicate. In SLANG they are called guards, and in FUNSOFT simply predicates. A transition is said to be enabled when its input places contain the sufficient quota of tokens and the predicate(s) on the transition is satisfied.

A transition along with its attached predicates and input and output places correspond to a process step, and is necessarily the minimal unit of commonality for Treaties, since in general altering the input or output places of a transition requires to modify the transition itself (analogous to changing the number or types of the formal parameters to a function in a conventional programming language). Also, the predicate is a local constraint on the transition and therefore conceptually part of it.

Integration of a process step into an existing net as part of the *import* operation involves: (1) merging (or adding new) output places of local steps with input places of the imported step; and (2) merging output places of the imported step with (possibly newly created) input places of local steps. This in turn might imply further modifications in the neighboring transitions to accommodate the changes in input-output places. These operations effectively merge the imported (common) step with the local process (net). It is not mandatory, however, to connect an imported step to the net. There might not be opportunities to do so, just as it is possible that in rule-based PMLs an imported rule will not match with any local rule, leaving it isolated, in which case there are no pre- and post-Summit actions during its enactment.

The Summit protocol starts when a common transition is attempted, and the input places contain some tokens representing remote objects (again, we assume remote binding capabilities which are provided by the underlying PSEE):

Summit initialization — The coordinating SubEnv binds the data arguments to its input places, and all involved SubEnvs mark their nets like the coordinating SubEnv, except the tokens in the non-coordinating SubEnvs are merely stubs.

Pre-Summit — The transition's predicate (if any) is evaluated at the coordinating site, and if not satisfied, the involved SubEnvs are notified. Since Petri net based PMLs are usually not extended to support the equivalent of backward chaining in rules, pre-Summit might be restricted to condition evaluation if needed to be performed in a distributed manner.

Summit — The transition is fired in the coordinating SubEnv, invoking an activity on the data arguments. When the activity finishes, all involved remote SubEnvs fire the transition *without executing the activity.* If there is a conditional branching that depends on the result of applying the activity, then the same "return code" is used in all SubEnvs to properly direct the flow of tokens to the output places.

Post-Summit — All associated SubEnvs transfer the appropriate tokens from their input to their output places. This can lead to firing of local transitions depending on the local nets. When local firing of transitions that were triggered by the Summit transition completes, the remote SubEnv notifies the coordinating SubEnv.

Summit-Completion — The coordinating SubEnv checks if new Summits can be derived from the previous Summit, based on further connections in the coordinating SubEnv's net. If none exist, the Summit is complete.

One way to look at a Treaty and a corresponding Summit in Petri nets is as an "intersection" subnet which is shared by the participating local nets (although possibly with different execution privileges), whereby each local net has its own private connections to the subnet, and its own "role" in the shared subnet, in terms of sending the data required for executing the Treaty subnet.

2.4.1.1 An Example

The following example, depicted in Figure 2.8, illustrates how Treaties and Summits can be applied in Petri nets. This is a multi-process extension of an example which was originally given in [11] describing SLANG.

In the example, there are two processes, CODE and TEST, used by two separate groups that are responsible for coding and testing the application, respectively. In order to increase productivity and consistency, the two teams, previously not connected in any way by their processes, decide to collaborate. The main collaborative step involves a joint evaluation of the test results by representatives from both groups that will lead to better understanding of the errors. In addition, implications of this step should provide local feedback to both groups. Finally, the necessary data transfer among

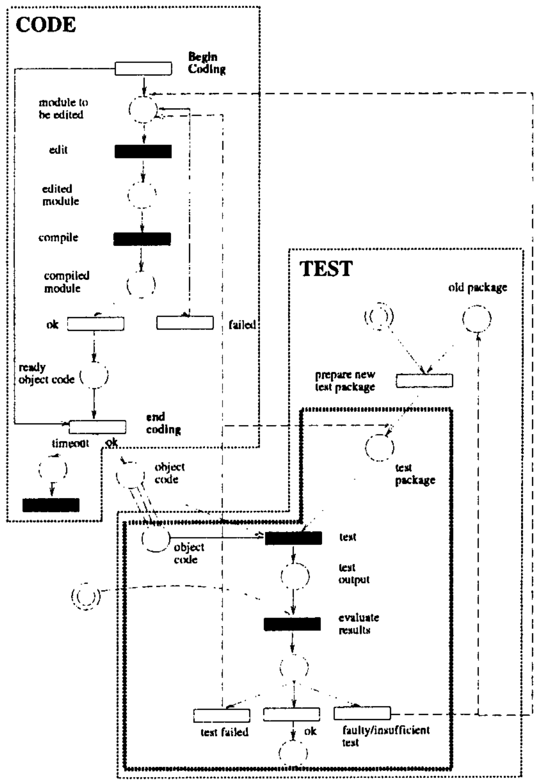


Figure 2.8: Example Multi-Process Petri-net

the groups (e.g., object code, reports, etc.), previously done outside the process, should be modeled and handled through the inter-process modeling and binding mechanisms, respectively, thereby enabling automatic and consistent transfer of the artifacts between the collaborating groups.

The dashed sub-process within the TEST process is then identified as the future shared sub-process. The main modifications made to that sub-process before turning it into a Treaty sub-process are in the addition of an interface input place (depicted by a circle with an inner-circle, representing in SLANG an end-user interacting with an activity) from the CODE group for purposes of the evaluation of the test results, and two new transitions with cross-process implications: (1) if the test fails, the CODE group is notified to fix the problems indicated by the test; (2) if the test is recognized as faulty, or insufficient, the TEST group is notified and modifies its package according to the recommendations made in the evaluation. Finally, the input place holding the object code is now transferred by the CODE group through the Summit mechanism, whereas before it was implicitly supplied to the TEST group. This, however, does not require a change in the sub-process, since when the Treaty is established, the object-code output place in the CODE process is merged with the corresponding input place in TEST.

Once the Treaty is established, all coding and test package preparations are still done independently and autonomously as before, but the processes synchronize for the actual testing phase when both groups are ready, as indicated by the presence of their respective tokens in the input places of the shared activities.

When the shared activities (i.e., the Summit) complete, a "fan-out" (or post-Summit) occurs, involving passing the relevant evaluation results to each team, possibly affecting their (local) state. At a later point, when both teams are ready for a second test, a second Summit activity is initiated.

2.4.2 Grammar-Based PMLs

The grammar hierarchy [41] and the corresponding automata provide another powerful set of formalisms for modeling a wide variety of systems, although they may have been less frequently applied to software process modeling than the other paradigms mentioned. There is a spectrum of approaches to employing grammars in process modeling, analogous to sentence generation at one end (what Heimbigner calls a prescriptive process [96]) to sentence recognition (parsing) at the other (proscriptive) [120]. The PDL project employed the former for context-free grammars [114], while the implementation of the Activity Structures Language on top of Marvel follows the latter approach [127]. One group experimented with both in the context of attribute grammars for HFSP [131] and Objective Attribute Grammars [206], respectively.

Considering the grammar-based PMLs, a *terminal symbol* corresponds to an activity in our context hierarchy, a *non-terminal symbol* to a task, and a *production* to a process step. Grammar-based PMLs usually associate some kind of condition with each production, or possibly with each symbol in a production, to specify when it could be selected. For example, in the PDL-based system these are called restriction conditions, in the Activity Structures Language they are simply rule conditions, and in HFSP they are decomposition conditions. Symbols are associated with formal and actual parameters in some fashion specific to the PML and PSEE. The symbol (along with its possible condition) seems the best candidate for the unit of commonality. But it doesn't have to be a terminal symbol. This reflects the hierarchical decomposition property of grammar-based PMLs, since it essentially allows to define any sub-process as common. However, any sub-tree that can possibly be generated during execution from that symbol must be identical in both processes (otherwise it will not be common). Thus, the *import* of a symbol is necessarily recursive, i.e., when a symbol is imported, all of its possible productions are imported recursively. Of course, a cyclic import must be detected as part of the *import* procedure.

As with Petri nets, the importing site must also explicitly augment its grammar with the new symbol, and use it in its production(s). An issue that comes up in all PMLs but is particularly eminent here is the issue of (sub)task naming. The newly imported symbol must not conflict with the name of any other local symbol, and at the same time it (and in fact all the derived symbols in a Treaty) must be identified as the common symbol when the Summit is enacted, eliminating simple local renaming as an option. The general approach recommended here, (which is the one actually taken in Oz to address naming of rules) consists of separation of logical and physical names combined with unique physical name generation. This approach enables both private (logical) naming of subtasks, as well as a global name space for running Summits. The Summit protocol works as follows (skip the first and last phases):

Pre-Summit — This phase begins when an activity represented by a common symbol is invoked in one process with data from multiple processes. The remote SubEnvs are notified, and any prerequisites to enacting that symbol are checked in each of the participating SubEnvs, each according to their own local process. In principle, a recognition-oriented PSEE might now recursively enact any symbols immediately preceding the common symbol in the current production in an attempt to fulfill the prerequisites, analogous to backward-chaining for rule-based PSEEs. This could be regarded as a form of sentence generation.

Summit — Assuming all SubEnvs ultimately agree, the symbol is enacted in the coordinating SubEnv. If, however, this is a non-terminal symbol representing a composite subtask, it is "parsed" recursively, possibly involving multiple multi-site activities. This is in fact a "natural" instance of composite Summits mentioned in the generic model. This is also why non-terminal Treaty symbols are imported recursively: a common sub-task must be literally common so that all involved sites know (and trust) what exactly is taking place when their data is accessed.

Post-Summit — All the participating SubEnvs are notified by the coordinator to complete the symbol. For example, in the case of a generation-oriented PSEE, each local process might automate control flow through its local production within which the symbol was embedded. Once again, the productions including a common symbol might be completely different in different local processes, and enacted independently and autonomously.

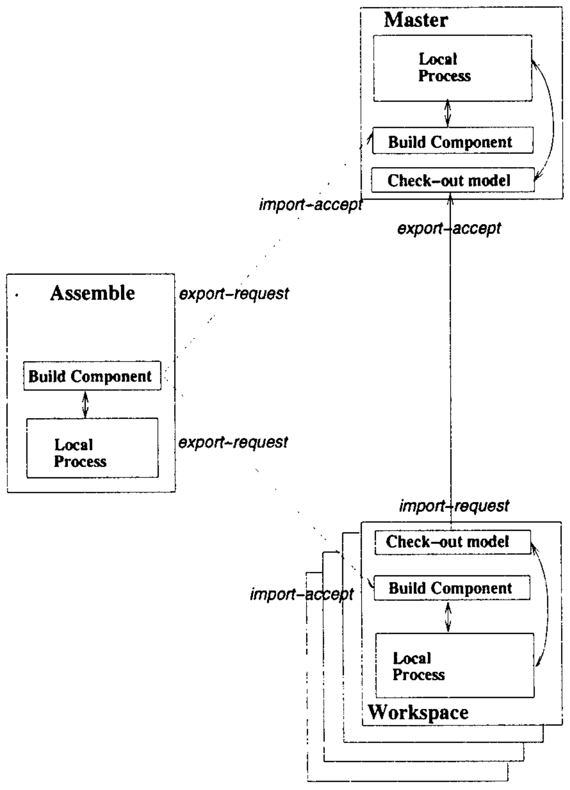


Figure 2.9: The Emerald City Environment

2.5 Experience and Evaluation

We now discuss how the interoperability model fulfills the requirements set forth in Section 2.1.1. We base our evaluation mainly on our experience in using Emerald City, an Oz environment that has been used to develop the Amber [224] rule processor, Pern [105] transaction manager, and the Darkover [144] object management system. Most importantly, Emerald City was used for the re-engineering of the Oz kernel itself to be constructed by these components[4].

2.5.1 The Emerald City Environment

Emerald City consists of three types of processes: a "Master" process that is used to maintain stable versions of components as well as additional "glue" modules that together comprise Oz; an intermediate "Assembly" process used for system (re)-engineering from components; and a "Workspace" process for individual intra-component development. Although processes can be in general instantiated for a variety of projects, Emerald City was tailored specifically to support the complex development and re-engineering tasks of Oz, so we will not distinguish from now between the processes and the actual environment. Note, however, that many local (i.e., non-Treaty) rules have been reused from earlier Oz and Marvel environments, particularly from OzMarvel, the Marvel environment which was used for the production of the earlier Oz 1.0.

Figure 2.9 shows the site interconnections in Emerald City. It comprises of a single Master SubEnv, a single Assembly SubEnv and multiple Workspace SubEnvs. The Workspace SubEnvs are mostly similar but not identical to each other, and unlike the multi-user Master and Assembly SubEnvs, they are mostly single-user although nothing prevents them from being used by mul-

[4]The latest version of Oz , 1.1.1, already uses Darkover and Pern as its components, and the integration of the Amber process-server is in progress.

tiple users (as some have). The Master SubEnv interoperates with Workspace SubEnvs via the `check-out-model` strategy that contains various cross-site rules for reserving and depositing artifacts across the sites, and for updating local information as a result of changes in other SubEnvs. A sample Treaty rule for updating function interfaces is listed in Appendix A. The rules in `check-out-model` are executable from Workspace, and have originated at Master, as indicated in the figure by the *request* and *export* labels, respectively.

Assembly maintains a three-way Treaty on the `build-component` strategy with the Master SubEnv and with each Workspace SubEnv that is involved in the re-engineering effort. This is a *simple* Treaty from Assembly to Master and Workspace, i.e., only the Assembly SubEnv can execute rules from that Treaty on data from Master and Assembly. A representative rule from `build-component` is listed in Figure 2.10.

This rule takes 3 arguments, one from Master, one form Assembly, and one from a Workspace SubEnv (line 1). It then binds the proper subsystem object from Assembly (line 5), the executable from the local workspace SubEnv (line 8), the local repository of header files (line 10), the main function from the local project (if exists) or from the master project (lines 12 - 15), and source files from the local and the master projects (lines 17 - 18). The condition (lines 20-24) states that all source files have been compiled and do not require recompilation (e.g., due to changes made to external function prototypes). Notice that this rule may backward-chain to a local `compile` rule vie the predicate in line 22. The activity (lines 25-28) invokes a builder tool with objects from all three sites. Finally, the effects (lines 29-31) indicate whether the build activity was successful (first effect) or not (second effect).

The Workspace SubEnv is where most of the development is done, where each developer tailors his/her own rules and tools to suit his/her needs. Thus, Emerald City is prescriptive and allows freedom in the creative aspects of programming (carried out in Workspaces) while providing automatic utilities and proscription for the complex and mechanical aspects of connecting the individual pieces together.

Emerald City has been in use since April 1995 and is constantly evolving. In its present configuration it consists of 16 SubEnvs: 1 Master, 1 Assembly, and 14 Workspace SubEnvs. The Master process consists of 34 local rules (in addition to the 15 standard rules used for site configuration [26] and for built-in operations, e.g., `copy object`) and 21 Treaty rules. Figure 2.11 shows a snapshot of user `kaiser` working in the Master SubEnv and listing the rules. The number of rules in the menu is smaller than the total number of rules because some of the rules are "hidden", i.e., they are intended to be fired only through chaining and not explicitly invoked by users, and other rules are overloaded, e.g., there are 6 different `reserve` rules for the various types of objects reserved, and for different types of destination SubEnvs.

The Assembly process has 27 local process-specific rules and 2 Treaty rules. A typical Workspace process contains 22 new rules, in addition to the imported Treaty strategies from Assembly and Master. Thus, 19% of the total distinct rules in Emerald City are Treaty rules. This figure, which can be used as a (static) measure of the level of site-interoperability, seems to be typical for Oz environments; in another experimental environment that implemented the ISPW9 "benchmark scenario" [176], this interoperability measure was 15% (see [21]). Another (dynamic) measure of site-interoperability is the percentage of actual invocations of Treaty rules in Summits from the total invocations. Table 2.5.1 summarizes the runtime statistics made for 11 active project members (taken from execution log files generated by Oz). The `built-in` column includes operations such as printing an object and browsing the hierarchy, and are in general not part of a specific process. The `treaty` column lists the "meta" administrator commands to establish/update/remove treaties. They were mostly issued by three administrators of Master and Assembly (other users have issued such commands sparsely, to connect their Workspace to other SubEnvs). The `summit` and `local` columns list invocations of Summit and local rules, respectively, and the `int. measure` lists the dynamic interoperability measure, i.e., the percentage of Summit rules from the sum of Summit and local invocations. We can see that the overall dynamic interoperability measure is 22%, meaning that approximately 80% of the development efforts were local, an overall positive result.

```
# -----------------------------------------------------------------
# Build a system using a local main function
#       ?lp is the local (workspace) project object
#       ?ap is the assembly project object
#       ?mp is the master project object, where code needed by a
#       variety of workspace SubEnvs is stored.
# -----------------------------------------------------------------
build-system [?lp:LOCAL_PROJECT, ?ap:PROJECT, ?mp:PROJECT]:
   # RULE BINDINGS:
  (and
    # find the proper subsystem in re-project to link to.
    (exists SUBSYSTEM ?s suchthat (and (ancestor [?ap ?s])
                                       (?s.Name = ?lp.subsystem)))
    # find the local binary
   (exists BIN   ?lb    suchthat (member  [?lp.bin  ?lb]))
   # find the repository of local header files
   (forall INC   ?i     suchthat (member  [?lp.inc  ?i]))
   # Find Main file, if it exists, in the local project or Master
   (exists CFILE ?main suchthat (or (and (linkto   [?lp.main  ?main])
                                         (ancestor [?lp ?main]))
                                    (and (linkto   [?mp.main  ?main])
                                         (ancestor [?mp ?main]))))
   # bind all source files in the local project
   (forall COMPILABLE ?c suchthat (or (member   [?lp.files ?c])
                                      (member   [?mp.files ?c]))))
  :
  # RULE CONDITION: all source files are compiled and are not marked
  #                  for recompilation
  (and no_forward (?c.compile_status = Compiled)
       no_chain   (?i.recompile_mod = false)
       no_forward (?main.compile_status = Compiled))

  # RULE ACTIVITY: build an executable with objects from all sites
  { COMBINE_TOOLS build_local_main ?lb.executable ?lp.build_log
                 ?s.libraries ?s.build_order ?s.med_libraries
                 ?main.object_code ?c.object_code }

  # RULE EFFECTS:
  (?lb.build_status = Built);      # build succeeded
  (?lb.build_status = NotBuilt);   # build failed
```

Figure 2.10: Three-site Build

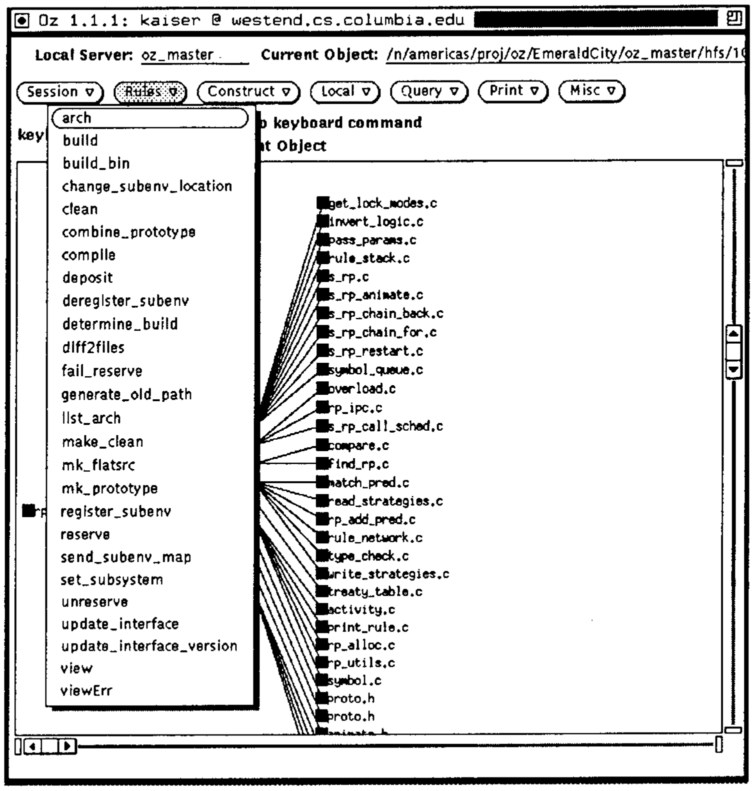


Figure 2.11: A Snapshot from the Master SubEnv

users	built-in	treaty	summit	local	Total	int. measure
1 admin	17141	661	2096	4908	24806	.30
2	2116	0	450	1145	3711	.28
3 admin	7707	134	589	1591	10021	.27
4	2812	29	735	1942	5518	.27
5	4661	23	330	1271	6285	.21
6	1250	0	110	446	1806	.20
7	631	13	133	519	1296	.20
8 admin	7049	198	678	3528	11453	.16
9	3968	1	294	1592	5855	.15
10	360	0	12	120	492	.09
11	11427	4	107	1782	13320	.06
Total:	59122	1063	5534	18844	84563	.22

Table 2.1: Summary of usage in Emerald City

2.5.2 Evaluation

2.5.2.1 Autonomy

Throughout the chapter we have seen numerous cases where autonomy played a major role in determining the design of the model and the system. Perhaps the major aspect that fulfills this requirement is that site autonomy is the default and is guaranteed unless explicit specification of interoperability is made. Autonomy-by-default is closely related to enabling independent operation, but includes also definitional and execution aspects.

Regarding definition, the schema, process, and database are all by default autonomous. The fine-grained modeling of Treaties contributes also to autonomy since each site can control precisely what is shared and what is not. The loose commitment to a Treaty that enables unilateral retraction further supports autonomy, even though it incurs some performance overhead in dynamically verifying Treaties at runtime. Regarding execution, the general idea in supporting autonomy was to minimize the impact of interoperability beyond what was explicitly defined as shared, and to maximize local execution. Most of these arguments hold equally well to the generic model as well as to Oz.

The tension between supporting autonomy and enabling facilities for interoperability have led to some oversights regarding autonomy, however, mostly in the design of Oz (as opposed to the generic model). The most important one concerns the global configuration mechanism and the global objectbase browsing facility, both of which cannot be "turned-off" and thus they violate autonomy. This was evidenced in Emerald City, where individuals working in their workspaces sites did not want to provide *any* access to other workspaces. To overcome this problem, the configuration mechanism has been modified to allow for partial visibility of remote sites which is determined autonomously, but a more general solution is needed.

2.5.2.2 Locality

To a large degree, this requirement was met, both in the generic model and in Oz. The model was specifically designed to minimize the impact on local work. In particular, the approach of gradually superimposing interoperability on top of the underlying (possibly pre-existing and enactable) local processes, maximizes locality. As far as the impact of decentralization on the quality and performance of local work — this issue seems to have been successfully met, too. The overhead imposed by Oz on local work in a SubEnv compared with work in an equivalent single instance running under the Marvel single-site PSEE is negligible, because the infrastructure overhead impacts only interoperability.

2.5.2.3 Interoperability

Given that autonomy was a crucial requirement, this "competing" requirement seems to also have been adequately addressed. The Treaty abstraction appears to support particularly well interoperability modeling of process and data. Two areas that still need improvements are in modeling interoperability at the user and the tool levels, which are related to groupware technology. Preliminary work has been done in [29].

Work in Emerald City revealed another area that requires improvements in Oz, namely better support for multi-site operations between trusted sites, particularly for interoperability modeling. The Treaty operation as a single command (with the issuer being administrator in both sites) was a step in that direction. Other improvements include commands for defining multi-site Treaties, more selective Treaty invalidation procedures that do not invalidate Treaties unnecessarily, and automatic updates of strategies without requiring to re-establish Treaties. Finally, work in Emerald City showed that establishing cross-site links at the data level is important for facilitating multi-site activities, although it may violate autonomy.

2.5.2.4 Support for Pre-existing and Heterogeneous Processes

Both Summits and Treaties were designed with this requirement in mind, and proved to be quite effective. It is of course possible and even likely that two pre-existing and unrelated processes will have no common sub-process a priori. But "bridges" of interoperability can be incrementally added, with minimal distractions to local work. This is particularly true for the declarative rule-based PML. For other PMLs, however, the addition of a new-subprocess may require more work and special tools. Another problematic issue with supporting pre-existing processes is with their *schemas*, particularly in strongly-typed PMLs. Such PMLs should provide facilities that enable to superimpose new shared sub-schemas on top of the pre-existing ones (perhaps along the lines of what is done in Pegasus [63]). Alternatively, PMLs might need to sacrifice some of their typing restrictions, at least for Summit activities, to accommodate heterogeneous schemas, and to be able to check for schema (sub)compatibility.

2.5.2.5 Scaleability

The Treaty/Summit model scales up mainly because it does not assume any global authority or centralized control. However, it does not provide means to form hierarchies over a set of interoperating sites, and they are all treated flatly as peers. This might have a negative impact on scaleability, particularly for top-down oriented environments. For example, in Emerald City it may have been advantageous to define a hierarchy of workspace SubEnvs with individual student's workspaces below "component" workspaces.

2.5.2.6 Language vs. System approaches to Treaty Definition

We already discussed in Section 2.2.2.2 some advantages of using the system-based approach. One disadvantage of this approach is that it is impossible to define a "Treaty" program with site classes as formal parameters. In Emerald City, for example, this capability would have allowed to automatically form a Treaty upon instantiation of a new workspace site in Emerald City. Instead, it was necessary to form a Treaty manually between each new workspace SubEnv and the other sites. Another disadvantage of the system-based approach was that it was necessary to create a set of built-in system calls not only for creating Treaties but also for removing, listing, and updating them. This suggests language constructs for Treaties in conjunction with system calls that are called from them, as an improved approach.

2.6 Related Work

ISTAR [57], one of the earliest software engineering environments (or "Integrated Project Support Environments"), provided comprehensive support to the software development lifecycle, including both management and software engineering. The main idea in ISTAR was the *contractual* approach, in which a "contractor" (e.g., a group of programmers) provides services to a client (e.g., a manager). The contract must have well-defined deliverables and acceptance criteria, and might include additional constraints imposed by the client. A contractor can further delegate some of the tasks to a sub-contractor, creating a "contract hierarchy" in a top-down fashion. In addition, the ISTAR architecture permits for sub-contracts (and all of their sub-contracts, recursively) to operate autonomously in different sites, since the contract databases are distinct and can be operated independently. Although ISTAR was not a PSEE (it had a somewhat hard-coded process), its architecture was an important step towards decentralization.

Shy, Taylor, and Osterweil were among the first to explicitly identify decentralization as a key environment technology [207]. Their theoretical work draws an analogy between software development and the business corporation, and they advocate a "federated decentralization" model for PSEEs with global support for environment infrastructure capabilities and local management with means to mediate relations between local processes. Among the arguments made for this model are: (1) The level of global support is not rigid; (2) While the communication is established under guidelines determined by the global process, the actual communication is provided and maintained under the control of the local entities; and (3) Extensibility, because integration of processes and services can be implemented gradually. This preliminary model, while advocating decentralization, still considers every sub-environment to be strongly affiliated with the corporation and necessarily abiding by some global rules. Thus, autonomy is necessarily restricted a priori.

Heimbigner argues in [97] that just like databases, "environments will move to looser, federated, architectures ... address inter-operability between partial-environments of varying degrees of openness". He also notes that part of the reason for not adopting this approach until recently was due to the inadequacy of existing software process technology. However, his focus is on support for multiple formalisms. His proposed ProcessWall [98] is an attempt to address heterogeneity at the language level. The main idea in the ProcessWall is the separation of process *state* from the *programs* that construct the state; in theory, multiple process formalisms (e.g., procedural and rule-based) can co-exist and be used for writing fragments of a process. However, decentralization as a concept is not addressed, and in particular, the process state server is centralized.

ProcessWEAVER is a commercial product of Cap Gemini Innovation, with a Petri net based PML. Fernström describes "...in a process, which consists of a set of cooperating sub-processes, every sub-process can be characterized by the set of 'services' it provides and requires from the other sub-processes" [67]. This sounds remarkably similar to our approach. However, in the ProcessWEAVER system, "...processes are recursively structured into sub-processes of finer and finer granularity and detail." In other words, processes are defined top-down, and provide essentially for fine-grained decomposition of one global process, whereas in our approach, what is in effect the decentralized process of a global environment can be defined bottom-up from the (collaborating) processes of the constituent SubEnvs. Finally, autonomy concerns for local process and their artifacts, which is a fundamental requirement in our approach, is not considered.

SMART [77] is an attempt to provide a methodology and a supporting technology for the *process* (as opposed to product) lifecycle through multi-formalism support, whereby different phases in the lifecycle are supported by different formalisms and corresponding (sub)systems. Specifically, SMART views the lifecycle of a process as consisting of a development phase; followed by analysis and possibly a simulation phase; followed by an embedding phase, in which a process model is instantiated with actual tools and product data bound to it; followed by an execution and monitoring phase, which feeds back to the development phase. Modeling, analysis, and simulation are performed with the Articulator system [160], process execution is performed by HP's SynerVision, and Matisse [78] (also from HP) is used to maintain a knowledge-base containing the artifacts that represent the process models developed in the Articulator, and serves as an integration medium between Articulator and SynerVision. Thus, the emphasis is on multi-paradigm support for the process, and on bi-directional translation: from process models to process (executable) programs, and from the process execution state back to the process model level. From a heterogeneity standpoint, SMART can be categorized as having some degree of system heterogeneity, since it integrates three different systems, and formalism heterogeneity, although not for defining different aspects of the process (as in ProcessWall), but rather for supporting different phases of a predefined lifecycle. However, there is no support for multiple processes with distinct instantiated products.

TEMPO [170] is another PSEE that is designed to support "programming-in-the-many", i.e., projects that involve a large number of people, and therefore its emphasis is on modeling and mechanisms for supporting collaboration, coordination, and synchronization between project participants. TEMPO provides three main abstractions that facilitate modeling multi-user aspects of the process: (1) hierarchical decomposition of processes to sub-processes in a top-down fashion, similar to ProcessWEAVER; (2) support for multiple private views of the process, through the *role* concept which allows to define private constraints and properties; and (3) active and programmable *connections* between role instances, which are defined and controlled by *rules* with temporal constraints in addition to pre- and post-conditions. TEMPO is data-centered, and is built on top of Adele 2 [20], an active configuration management system with data-driven triggering, which enables to realize rule processing in TEMPO. While TEMPO provides for definition of "personal" processes and supports coordination among them, it is still inherently centralized, in that it requires a single database as the coordination platform, and supports multiple views of essentially a single group process, defined in a top-down fashion.

2.7 Conclusions and Future Work

Two key concerns guided this research: (1) maximizing local autonomy, both physically and logically, so as not to force a priori any global or inter-site constraints on the definition, execution and operation of local sites, unless explicitly specified in a particular environment instance; and (2) flexibility and fine-grained control over the degree of interoperability.

The high-level approach to address decentralization was to extend the notions of process modeling and process enactment to inter-process modeling and inter-process enactment, respectively. The former was achieved by the *Treaty.* In essence, a Treaty is an abstraction that specifies shared sub-processes for interoperability purposes while retaining the privacy of the local sub-processes. Treaties have several unique characteristics. First, they require explicit and active participation of the involved entities to mutually agree on the nature of the interoperability, thereby balancing autonomy and global specification. Second, the definition of Treaties is fine-grained in two respects: they are defined pairwise, between every two sites that need to interoperate, as opposed to being global and known in all sites of a multi-site environment; and each Treaty is formed over a single and a small sub-process unit. Still, complex Treaties can be formed (and subsequently executed) between any number of sites and involve arbitrarily large sub-processes, by successive invocations of simple Treaties (which could be optimized from the user interface perspective). The third characteristic of Treaties is that they are superimposed on top of pre-existing processes as opposed to being specified as part of each individual process; this enables gradual and incremental establishment of interoperability and supports the decentralized bottom-up approach. Fourth, they are designed to support local evolutions including unilateral retraction from Treaties (combined with dynamic Treaty verification), on demand.

Inter-process execution was achieved by the complementary *Summit* model. Summits are the execution abstraction for Treaty-defined sub-processes. They support multi-site enactment of shared sub-processes involving artifacts and/or users from multiple sites, while maximizing local execution of related private sub-processes. This is done by successively alternating between shared and private execution modes: the former is used for the synchronous execution of the (fine-grained) shared activities, involving artifacts, tools, and/or users from multiple sites, and the latter is used for the autonomous execution of any private subtasks emanating from prerequisites and consequences of the shared activities.

2.7.1 Future Work

The first issue to further explore is extensions of the basic Treaty/Summit model with more abstractions that support alternative modes of interoperability, both in modeling and in execution. For example, enabling to model and enact local activities that execute simultaneously. Another extension concerns enhanced groupware modeling facilities for tools and users. Finally, support for site hierarchy should be explored.

Addressing heterogeneity and interoperability at the PSEE and PML levels in conjunction with the process-interoperability model described in this paper, are other important avenues to explore.

Finally, it seems that the idea of describing the behavior of autonomous entities formally, as a basis for constructing consistent and trustworthy interoperability among them, and operating within an environment that supports their execution, goes beyond software process modeling and can be applied to general distributed and decentralized system design. For example, this could be used to model and subsequently support interoperability among autonomous Internet repositories, making them more active and responsive to other objects on the network.

Appendix A: A Sample Treaty Rule from Emerald City

```
# -------------------------------------------------------------------
# A Treaty rule for Updating interfaces for Workspace Project
# (LOCAL_PROJECT) based on changes in the Master Project (PROJECT)
# -------------------------------------------------------------------

update_interface[?lp:LOCAL_PROJECT, ?p:PROJECT]:
  (and (exists SUBSYSTEM ?s  suchthat (and (ancestor [?p ?s])
                                          (?s.Name = ?lp.subsystem)))

       # Find local interface
       # --------------------
       (forall PROTOTYPE ?LPT suchthat (member   [?lp.proto    ?LPT]))
       (exists INC       ?i   suchthat (member   [?lp.inc      ?i]))
       (forall HFILE     ?h   suchthat (member   [?i.hfiles    ?h]))
       # Find master interfaces
       # ----------------------
       (forall PROTOTYPE ?PT suchthat (member   [?p.proto ?PT]))
       (exists INC       ?ii suchthat (member   [?lp.interface ?ii]))
       (forall COMPONENT ?CD suchthat (ancestor [?s            ?CD]))
       (forall LIB       ?l  suchthat (linkto   [?CD.lib       ?l]))
       (forall MODULE    ?m  suchthat (linkto   [?m.library    ?l]))
       (forall SRC       ?sr suchthat (member   [?sr.libs      ?l]))
       (forall INC       ?ri suchthat
                              (or (member [?sr.incs       ?ri])
                              (linkto [?m.related_incs ?ri])
                              (member [?p.common_incs  ?ri]))))
  :

  { TREATY_TOOLS install_interface ?i ?h.contents ?ii ?ri.directory
                 ?sr.sys_includes ?p.tags ?lp.tags
                 ?PT.contents ?LPT.contents
                 return ?i_path ?ii_path ?combine }

  (and no_chain (?lp.interface_version = 0)
                (?lp.interface_version = ?p.interface_version)

       no_chain (?lp.sys_includes = ?combine)
       no_chain (?i.directory = ?i_path)
       no_chain (?i.recompile_mod = false)

       no_chain (?ii.directory = ?ii_path)
       no_chain (?ii.recompile_mod = false));
```

Chapter 3

Process Support for Componentry

Abstract

Componentization is an important, emerging approach to software development whereby new systems are constructed from relatively large-scale components intended to be used in a variety of systems. More significantly from the perspective of this report, componentization also provides a road to modernization of stovepipe systems, which are restructured into components to ease continued maintenance. Selected components in the original system can be completely replaced, e.g., the database or user interface, potentially in a family of configurations each including different realizations of the components. Of course, the newly separated components can also be reused in other systems.

We have investigated process modeling and workflow automation technology to support both reengineering of legacy systems into components and replacement of some of those components. This chapter describes our experience following that approach through two generations of component-oriented process models supported by two generations of process-centered environments developed in our lab.

3.1 Introduction

Componentization, as discussed here, has two main facets: *re-structuring* a stovepipe system into components that could potentially be reused in other systems, and *re-engineering* the original system to permit replacement of selected components. One application is to upgrade portions of the stovepipe system to new technology, e.g., a new database or user interface. Another is to migrate to a new architecture, for instance, converting a monolithic system to the client/server paradigm, with some old components appropriately encapsulated to work together with some new ones.

In this chapter, we describe our experience using process/workflow technology to support both aspects of componentization. Our study was particularly targeted to process support for componentization of process-centered environments (PCEs), which involved breaking up a particular existing PCE into components and reusing some of its components in a variety of environment architectures and frameworks, but the same processes should apply to other kinds of systems. We started implementing the existing PCE in January 1987, initially known as MARVEL and later as OZ; the final version consists of about 300k lines of C, lex and yacc code. Approximately 60 graduate and undergraduate students participated in the effort, most of them for only one semester as part of an independent study project for academic credit, a handful for several years as research assistants. An "exploratory programming" style was used, and the resulting PCE certainly qualifies as a stovepipe system.

We developed a series of two enactable (executable) process models, each intended to support both aspects of componentization, i.e., re-structuring and re-engineering, in an *incremental* manner. That is, the componentization effort was going on at the same time on the same code as "new" development work supported by the same process; it was not possible to suspend other changes to the system while the re-structuring/re-engineering was in progress. The first process, OzMarvel, was implemented in Spring 1993 as a MARVEL environment instance and then employed for nearly two years in our initial development of OZ. EmeraldCity came on-line in Spring 1995 as an OZ environment instance and has been used for two years as of this writing. (Prior to OzMarvel we used a non-component process called CMarvel, starting in January 1992, which mimicked our earlier work methods on Unix.)

There were two main reasons for upgrading from OzMarvel to EmeraldCity. One was to bootstrap from MARVEL to OZ as our platform to continue development of OZ. The OZ project is devoted in large part to componentization issues, while the predecessor MARVEL project was not. Another important distinction between MARVEL and OZ, for the purposes of this chapter of the report, is that a MARVEL environment instance supports a single process that must be enacted by all users of that environment, although they would generally follow distinct workflows, whereas an OZ environment instance supports interoperability among multiple autonomously developed processes and interactions among users carrying out workflows within different processes. A MARVEL environment with an in-progress process can be converted to a single-process OZ environment, but as explained later EmeraldCity needed to exploit OZ's multi-process support.

EmeraldCity is also substantially different from OzMarvel in several other dimensions due to our early experience using OzMarvel to divide a legacy system into components and integrate experimental systems from those and external components. Thus the second reason was to incorporate what we had learned from our initial, relatively naive attempt at a component-oriented process and continue our re-engineering work with the significantly better process (i.e., from the viewpoint of successfully supporting componentization).

It may be confusing that we used a PCE to support componentization of that same PCE. There is nothing specific to PCEs in either of our two software engineering processes, so the approach should apply equally well to other systems — but a PCE happened to be the system we were componentizing and from which our experience is drawn. That is, this was our real work, not an invented "case study". Certain peculiarities of the processes are, however, specific to C programming, e.g., the distinction between source and header files, the use of prototypes, etc.; we assume throughout the report that the reader is generally familiar with ANSI C.

First we provide brief background on the MARVEL and OZ process modeling and workflow automation systems. Then we describe the OzMarvel and EmeraldCity processes, including the requirements they were intended to fulfill, how they exploited the then-available process/workflow technology, and our experience using each of the process environment instances in our componentization efforts. The chapter concludes by summarizing lessons learned.

3.2 Marvel and Oz Background

MARVEL [31, 108] and OZ [27, 21] employ client/server architectures. Clients provide the graphical user interface and invoke external tools. Servers context-switch among multiple clients, and include the workflow automation engine, object management, and transaction management for concurrency control and failure recovery (transaction management details are not addressed in this chapter).

MARVEL and OZ employ nearly the same rule-based process modeling language in which to define new processes or tailor reusable processes for an organization or project. A rule generally corresponds to an individual software development task, and specifies the task's name as it appears in a user menu. A rule definition includes: typed parameters and bindings of local variables to the results of queries on the project objectbase; a condition on the parameters and local variables that must be satisfied before initiating the activity — generally an external tool invocation — to be performed during the task; the tool envelope and arguments for that activity; and a set of effects, one of which asserts the actual results of completing the activity on the objects referred to in the parameters and variables. There is generally more than one possible effect if the tool has more than one possible result (the simplest example is a compiler than generates either object code or syntax error messages).

The workflow engine enforces that rule conditions are satisfied, and automates workflows via forward and backward chaining. When a user requests to perform a task whose condition is not currently satisfied, the system automatically backward chains to attempt to execute other rules whose effects may satisfy the condition; if all possibilities become exhausted, the user is informed that it is not possible to enact the chosen task at this time. When a rule's activity completes, its asserted effect triggers automatic enactment of other rules whose conditions have now become satisfied. Both backward and forward chaining procedures operate recursively. Users usually control process performance by selecting a rule representing an entry point into a composite task consisting of a one main rule and a small number of other auxiliary rules (reached via chaining) to propagate changes and perform bookkeeping chores, but it is possible to define an entire process as a single goal-driven or event-driven chain — which is useful for simulation or training purposes. Built-in operations such as add an object, delete an object, etc., are modeled as rules for a uniform approach, and different conditions and effects can be attached to such operations for different classes of objects. OZ provides means for modeling and launching synchronous and asynchronous "groupware" tools [227, 29], which were not supported by MARVEL, but the details aren't relevant to this chapter.

MARVEL and OZ support nearly the same object-oriented data definition and query languages. A class specifies primitive attributes (integers, strings, timestamps, etc.), file attributes (pathnames to files in an intentionally opaque "hidden file system" that should not be accessed except through the PCE), composite attributes in an aggregation hierarchy, and reference attributes allowing arbitrary 1-to-N links among objects, and one or more superclasses from which it inherits attributes (and rules treated as multi-methods [15]). Ad hoc and embedded (in rules) queries may combine navigational and associative clauses in a declarative style. Rules perform all data manipulation in the objectbase proper, whereas the contents of file attributes are manipulated by tools. Commercial off-the-shelf tools and other external application programs are interfaced to an environment instance through shell script envelopes, using augmented notation that hides from tool integrators the details of accessing the "hidden file system" and passing input and output parameters [86]. A return code from the envelope determines which of the several rule effects is asserted.

A MARVEL environment consists of an arbitrary number of clients connected via an interprocess communication layer to a central server. Each server enacts one process, in which all its clients participate; an arbitrary number of workflows within that process may be in progress simultaneously, one or more per client. Every client maintains its own objectbase "image" for browsing, which includes composite and reference attributes without primitive attributes and files, which are transferred only as needed — so the "image" is relatively small.

In contrast, an Oz environment consists of one or more servers (termed "sites"), each with its own process model, data schema, objectbase and tools. Clients are always connected to their one "local" server (usually on the same local area network sharing a network file system), and may also open and close connections on user command to "remote" servers (which may, but need not, reside in other Internet domains with no shared file system). Oz servers communicate among themselves to establish *Treaties* — agreed-upon shared subprocesses automatically added on to each affected local process, and to coordinate *Summits* — enactment of Treaty-defined workflows that involve data and/or local clients from multiple sites. We stretch the International Alliance metaphor a bit, since Treaties among sites precede and specify Summits rather than vice versa.

3.3 First Try: OzMarvel

OzMarvel was our first component-oriented process. We used it to assist us in pulling subsystems out of Marvel to rewrite them into components, and at the same time implement Oz by direct extensions to Marvel — thus the name OzMarvel (our document processing environment was named DocMarvel, etc.). The main components envisioned were the process engine, the transaction manager, and the object management system. The long-term plan was to eventually reconstitute Oz from these components (the final phase, replacement of the process engine, is now in progress using EmeraldCity), after performing a set of experiments concerned with integrating some of our components into externally developed systems (notably Cap Gemini's ProcessWEAVER process-centered environment framework [67] and University of Wisconsin's Exodus database management system [38]) and replacing portions of Oz with externally developed components (notably the object management system from GIE Emeraude's PCTE industry-standard environment framework [221]); these experiments are discussed in [105, 183, 107, 147].

The OzMarvel data schema structures the objectbase into two main parts. One part consists of a set of teams, each consisting in turn of a group of private programmer workspaces. We used only two teams, representing current and past project members, respectively, but the schema allows for an arbitrary number. A workspace contains a set of C source and header files reserved by that user, locally generated object code and executables, and references to libraries in a shared repository needed to compile and build local executables. There are also means for testing with executables from other private workspaces (e.g., one user might be working on a new client while another works on a new server that must be tested together due to a change in the client/server protocol).

The other part of the objectbase consists of a set of projects, each representing a shared code repository. We had three, representing the baseline versions of Oz and its main Oz components, work progressing independently of Oz, and a frozen copy of all code delivered to a funding agency. The first two projects are collectively referred to as the "Master Area". Each project consists of a set of what the schema calls systems, a component pool, a module pool, and a pool of external libraries. Each system consists of a set of subsystems, each in turn corresponding to a distinct executable (a distributed system may involve multiple cooperating executable programs). For example, at the time we migrated the Oz code out of OzMarvel it had 19 subsystems: three variants of the server, four kinds of client, three translators for different Oz notations, the schema/process evolution tool, the daemon for automatically bringing up the server when a client starts up, and several utilities for managing an environment instance.

Libraries represent object code archives (i.e., Unix ".a" files) that may be linked into subsystems or components, together with their header files needed for compilation of importing code. For instance, OzMarvel had external libraries for `gdbm` (used as the backend of Oz's native object management system), for the PCTE object management system (which replaced `gdbm` in the two variant Oz servers), and for the `socks` secure TCP/IP sockets package (for authorized tunneling through corporate "firewalls"), along with `motif`, `xview`, `termcap`, etc. libraries imported by particular

Oz clients.

Each subsystem referenced the several *context-free* components and external libraries (in the component and library pools, respectively) from which it was constructed, plus special-purpose modules for "glueing" those components and libraries together to construct the specific subsystem. The components in turn referenced the *context-free* modules (in the module pool) from which they were composed, and also contained local "glue" files for tailoring its modules to provide the functionality needed for that component. Each module (which could be and often was decomposed into a hierarchy of submodules) contained attributes representing its source files, object code archive, and public (to using modules) and private (for use only within the module) header files, as well as references to other header files needed for compilation. We emphasize context-free here, meaning that the components, modules, etc. were not supposed to make any assumptions about the systems and subsystems in which they were to be used and, at least in principle, were amenable to "plug-n-play".

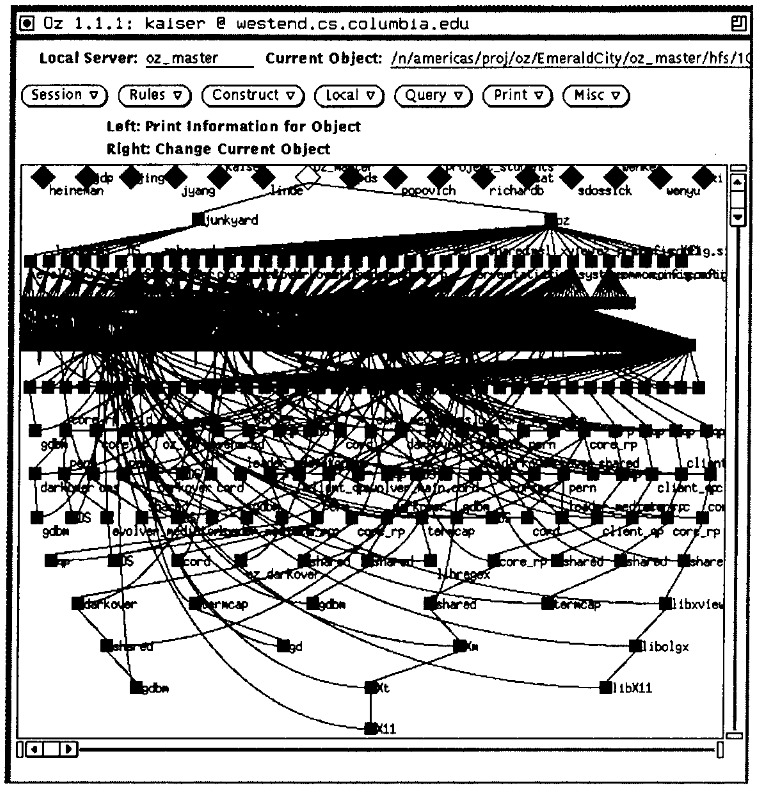
Over its lifetime, OzMarvel was actively used by 14 people (not all at the same time). Although the basic philosophy and design remained the same, OzMarvel was modified several times to fix bugs in the process and to improve multi-user support; see [25] for a brief discussion of the schema and process evolution utility, called Evolver, used by MARVEL and Oz to upgrade the state of an in-progress process to match the semantic constraints of a new process model. The final process evolution left OzMarvel with 139 rules (only 26 task names appeared in the user menu, due in part to overloading — for instance, the `edit` command applies to many different kinds of objects — and in part to the marking of 75 rules in the process model as for internal propagation purposes only); 48 classes (13 of them virtual superclasses, such as `VERSIONABLE`, which would never be instantiated); and 37 tool envelopes.

Unfortunately, OzMarvel's multi-level structure proved much too complicated, evidenced by the relatively large proportion of propagation rules needed. For example, OzMarvel's rules to automate maintenance of each source file object's set of references to the objects representing each of the directly or transitively included header files were particularly intricate (and buggy). A header file might include other header files with arbitrary recursion depth, and propagation rules were triggered whenever the source file or one of the header files was edited in a way that affected header file inclusion. Some of these references were different for each component/subsystem context in which the source file was used, since the same component might provide a different interface to different subsystems.

Further, multiple modules performed the same function with intentionally the same interface, i.e., there were at least two each of the major modules of the process engine, transaction manager and object management system in the module pool, corresponding to the original native modules in Oz vs. our new components. Thus the tools we had used for code cross-referencing in the earlier CMarvel environment, standard Unix **`etags`** and our home-grown **`revtags`**, which assume a flat name space, did not operate properly in OzMarvel. Renaming solved this problem, e.g., *component-name_subroutine-name*, but made it difficult to plug-replace one component with another since code had to be edited (or preprocessed) for each subsystem context.

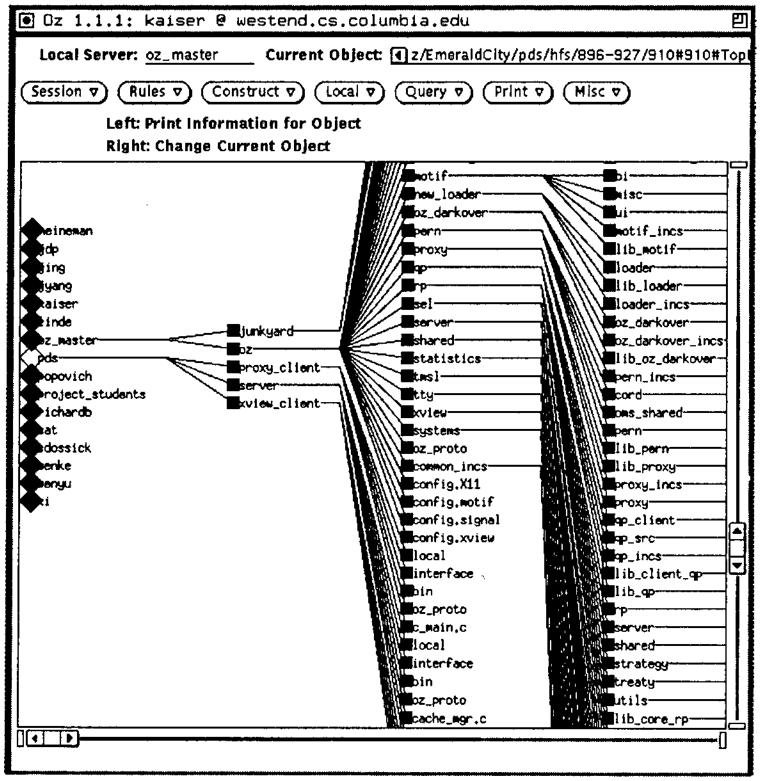
3.4 Second Try: EmeraldCity

EmeraldCity is really a set of several processes that work together, following Oz's International Alliance metaphor, rather than a single process like OzMarvel. EmeraldCity consists of two shared sites and an arbitrary number of workspace sites (16 at present, the number varies as students join and leave the project — or clone their workspace to perform relatively independent development in each one). Each EmeraldCity workspace has its own objectbase and process, whereas all OzMarvel workspaces necessarily are part of the same objectbase and share the same process. Thus moving to EmeraldCity gains advantages in performance (transfer of smaller objectbase images) and fault-tolerance (no central point of failure). EmeraldCity workspaces can be (and have been) shared by multiple users, but usually they are personal. One of the shared sites corresponds to the "Master Area" in OzMarvel, whereas the other "Assembly Area" is used only while a major re-engineering



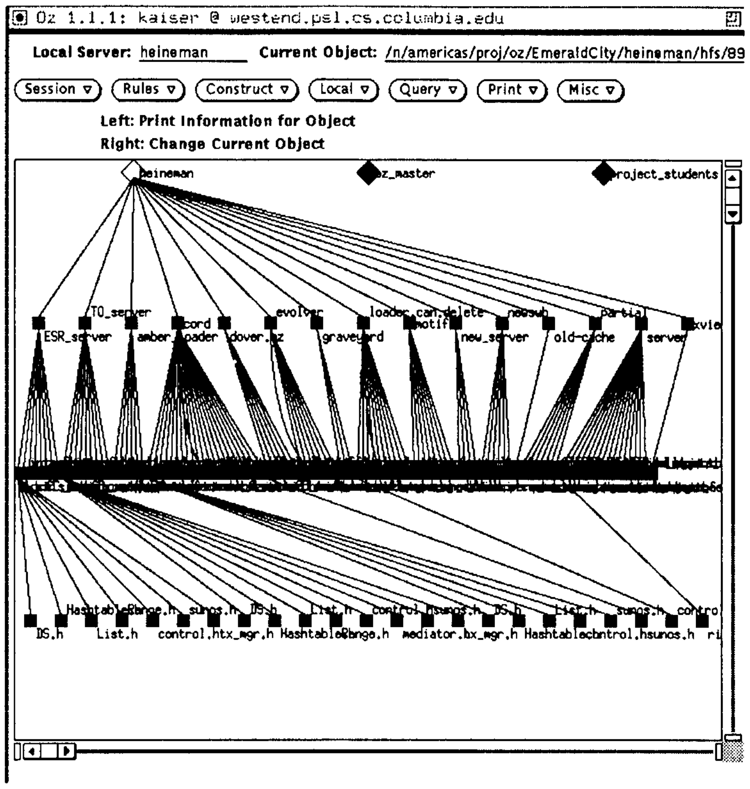
Oz 1.1.1: kaiser @ westend.cs.columbia.edu
Local Server: oz_master
Current Object: /n/americas/proj/oz/EmeraldCity/oz_master/hfs/1
Session
Rules
Construct
Local
Query
Print
Misc
Left: Print Information for Object
Right: Change Current Object
junkyard
oz
X11
libX11
Xt
libolgx
libxview

Figure 3.1: Hierarchical Master Area Display



Oz 1.1.1: kaiser @ westend.cs.columbia.edu
Local Server: oz_master
Current Object: z/EmeraldCity/pds/hfs/896-927/910#910#Top
Session
Rules
Construct
Local
Query
Print
Misc
Left: Print Information for Object
Right: Change Current Object
junkyard
oz
proxy_client
server
xview_client

Figure 3.2: Horizontal Master Area and Workspace Display


Oz 1.1.1: kaiser @ westend.psl.cs.columbia.edu

Local Server: heineman   Current Object: /n/americas/proj/oz/EmeraldCity/heineman/hfs/89

Left: Print Information for Object

Right: Change Current Object

Figure 3.3: Workspace Display

effort is in progress. Figure 3.1 shows the (not terribly readable) hierarchical view from the "Master Area" site (oz_master), showing only the local objectbase. Figure 3.2 shows a (somewhat more readable) horizontal view from that site, with an open connection to the pds site (Peter Skopp's workspace).

Over the summer 1995 we converted Oz from its native pointer-based object management system to using a OID-based object-oriented database component, Darkover [144]. An OID (or object identifier) is a unique identifier represented as an integer. The native transaction manager had already been replaced with a component [105], a much simpler effort performed using OzMarvel, and work on the new process engine component was still progressing independently.

Source and header file objects slated for re-engineering were checked out of the Master Area into a workspace (via a Summit) for changes, and then checked into either the Assembly Area (another Summit). Conversion efforts in relevant workspaces ranged from using home-grown tools that semi-automated the lexical aspects of interface changes by matching code patterns that should be replaced with calls to Darkover's application programming interface; to recoding individual subroutines to traverse OID arrays rather than linked lists pointing to child objects; to module redesign, e.g., of Oz's cache manager for remote objects, which involved modifying Darkover to support transient objects — which probably would not have been possible had we been integrating with a foreign object management component.

The Assembly Area process allowed only completely converted code to be checked in. However, other code could still be checked into the Master Area, permitting unrelated development of portions of the system. This was very useful since not all of the developers were involved in the re-engineering effort, but had other pressing work to do that we wished to disrupt as little as possible. Figure 3.3 shows the hierarchical view from the heineman site, which allows opening of only the "Master Area" and "Assembly Area" (called proj_students for obscure historical reasons).

Subsystem builds in each re-engineering workspace looked for non-local object code first in the

Assembly Area and, only if not found there, in the Master Area; other workspaces were unaware of the Assembly Area. Treaties between Oz sites are set up on a pairwise basis that is neither symmetric nor transitive, so the connection graph need not be complete, although Summits can involve any number of sites that have agreed to the same Treaty. Our goal was to always be able to perform recompilation and build throughout the three months or so while the 150k affected lines (out of about 280k) were converted; this incremental approach would not have been possible without the binary compatibility of the old and new interfaces, due to C's allowance of type casting between integers and pointers. After the re-engineering effort was over, the entire code base was copied from the Assembly Area to the Master Area.

We have also been incrementally re-engineering our previously Kernighan and Ritchie C code base to ANSI-standard, the main focus of a previous paper [103]. Although we settled on a mechanism somewhat more complicated than what was presented there, the effort has proved considerably simpler than the object management system replacement. After an initial flurry to convert the Master Area baseline, the work has proceeded more gradually: any C code (from OzMarvel or elsewhere) may be immigrated into an EmeraldCity workspace using a utility [213]. There it is converted using a combination of the Gnu **protoize** tool, extensions to the envelopes of other tools, and manual changes to header files to preserve the conventions discussed below. The Master Area enforces that only ANSI-compliant code can be deposited (i.e., the ANSI C compiler using the strictest options generates no error or warning messages).

EmeraldCity restricts the contents of header files to avoid transitive dependencies, simplifies OzMarvel's notion of pools, and distinguishes *context-free* from *context-sensitive* representations of components. A project is composed only of a set of systems, a single prototype header file (`oz-proto.h`) included by all other header files in the project, a set of header files containing type definitions (but no prototypes) that may be used throughout the project, and a "program unit" pool. A prototype is essentially a forward declaration of a C function signature, as it must be used in source files whose object code will link with that function's code; prototypes are a required feature of ANSI C. `oz-proto.h` is automatically constructed by concatenating the contents of the `proto.h` header file associated with every library, which are in turn constructed automatically by tool envelopes as files are edited and compiled and libraries archived. Sections of `oz-proto.h` are guarded with preprocessor variables, so that only the relevant subset of the prototypes are used during compilation and there are no naming conflicts.

An EmeraldCity system consists of only a set of subsystems, as illustrated in Figure 3.4. A subsystem consists of a set of *context-specific* components, an executable, an archive for "glue" code between the components, and the source and object code for the "main" file (required by C convention for every executable). Each *context-sensitive* component contains source files for "glue" code, a reference to the single library representing the entire component, and a hierarchy of subcomponents. These components are not reused in multiple subsystems — thus the designation *context-sensitive.*

In contrast, an EmeraldCity program unit is analogous to a *context-free* component in OzMarvel: it consists of a set of modules, a set of libraries, a set of local header files, and a set of references to header files in other program units. Each module contains its source files, and references the one archive holding its object code and the appropriate header files from its program unit; the modules assume the context of their program unit, but not of a subsystem component. Each workspace consists of a set of "local projects", which organize checked out files according to subsystem contexts and replicate relevant header files.

This new organization solved the naming difficulties that permeated OzMarvel. The files scanned by cross-referencing tools, `etags` and `revtags`, are always encapsulated in the appropriate context. We recently added a home-grown tool, called Hi-C, to generate HTML (HyperText Markup Language) to enable EmeraldCity users to view code and follow automatically generated hypertext links using World Wide Web browsers.

EmeraldCity has been actively used by about 15 people (not all at the same time). The Master Area site consists of 78 rules (26 distinct names visible in the task menu, coincidentally 26 rules for internal propagation), 27 classes (6 of them virtual), and 32 tool envelopes. 21 of these rules are

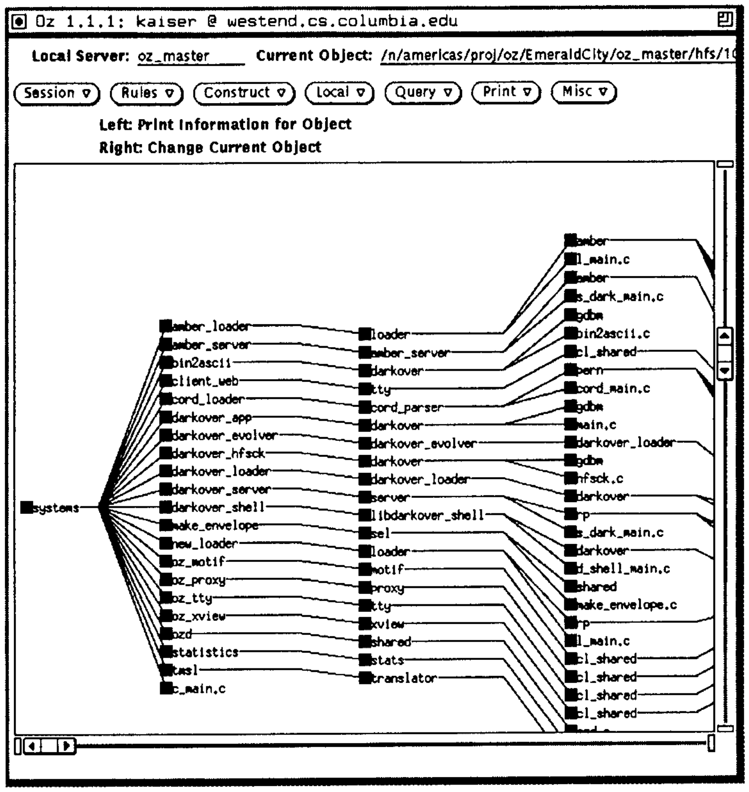


Figure 3.4: Zoom Into Systems Hierarchy in Master Area

exported via Treaties to workspaces for use in checkin/checkout, local build, etc. Workspaces are virtually identical to each other, although they need not be, with the main customization in the past being whether or not they formed Treaties (which have now been revoked) with the Assembly Area. A typical workspace has 68 rules (24 task names, 19 propagation), the identical 27 classes as the Master Area (this is not a requirement of Oz: different sites may have different schemas, with compatible subschemas needed only to match any Treaty subprocesses), and (coincidentally) 27 envelopes. The Assembly Area is the same as a Workspace, except for three special rules that were used in the Darkover conversion and two rules exported for the (now-revoked) three-way Treaties with a re-engineering workspace and the Master Area. One of the former three rules is shown in Figure 3.5 and one of the latter two in Figure 3.6, both in the appendix; the other rules are similar.

3.5 Conclusions

We originally imagined we would construct EmeraldCity by evolving OzMarvel, but that proved too complicated, so we designed the new process from scratch (although portions of OzMarvel's data model were retained). Between launching of OzMarvel and completion of immigration into EmeraldCity our code base nearly doubled from 155k to 280k lines; this does not include any external libraries or systems, e.g., for X Windows or used in our integration experiments. After that we consolidated and replaced code, with relatively little growth in the Master Area. [1] While neither MARVEL nor OZ are production-quality in the commercial sense, we have been using the technology on a daily basis for over five years and have licensed the technology to about 45 institutions.

Our reconstruction of OZ from components and experiments integrating components with/from

[1]Until we started developing OzWeb, an extension of Oz that operates on World Wide Web entities, discussed elsewhere [122].

commercial systems necessitated our development and use of two generations of component-oriented process models running on two generations of process modeling and workflow automation systems. Our processes focused on the nitty-gritty but mandatory details of code understanding and configuration management, and ignore upstream aspects of the lifecycle (which were performed off-line). Although some of the problems encountered in OzMarvel were due to peculiarities of C, we'd expect to run into analogous difficulties using most programming languages — given that few were designed with "plug-n-play" componentry in mind. Oz's support for process interoperability (Treaty and Summit) proved an immense boon to component-based software engineering of our process-centered environment framework and we expect would apply similarly to other legacy systems. The incremental nature of the component-based processes, and the corresponding support from our process/workflow technology, were critical in being able to perform re-structuring and re-engineering without significantly interfering with "new" development work affecting the same code.

```
# rule signature
convert_CLASS [?c:COMPILABLE, ?cf:PROTOTYPE]:

  # bindings of local variables to results of objectbase queries
  (and (exists LOCAL_PROJECT ?lp suchthat no_chain (ancestor [?lp ?c]))
       # Use local version of  prototype file
       (forall PROTOTYPE     ?LPT suchthat no_chain (member [?lp.proto ?LPT]))
       # Local HFILEs
       (forall INC         ?li suchthat no_chain (member [?lp.inc ?li]))
       # Installed Interface (from set_subsystem[] rule)
       (forall INC         ?ii suchthat no_chain (member [?lp.interface ?ii])))
  :
  # condition
  # If the C file has not yet been compiled, this rule can fire.
  # The compilation changes the status of the CFILE to Compiled on success.
  (and no_forward  (?ii.recompile_mod = false)
       no_chain    (?cf.Name = "CLASS_PTR")
       no_forward  (?li.recompile_mod = false))

  # activity
   CONVERSION_TOOLS converter ?c.contents ?c.compile_log ?c.object_code
                           ?c.proto ?li.directory ?ii.directory ?LPT.contents
                   "-DMOVING_CLASS -Wall" ?lp.sys_includes
                           ?lp.compiler_directives ?cf.contents

  # success and failure effects
  (and          (?c.compile_status = Compiled)
       no_chain (?c.object_time_stamp = CurrentTime));
  (?c.compile_status = ErrorCompiled);
```

Figure 3.5: Darkover Conversion Process Task

```
# Build with master Main file (in SUBSYSTEM or SYSTEM.common_main)
# rule signature
build[?lp:LOCAL_PROJECT, ?p:PROJECT, ?mp:LOCAL_PROJECT]:
  # bindings of local variables to results of objectbase queries
  (and (exists SUBSYSTEM ?s  suchthat (and no_chain (ancestor [?p ?s])
                                          no_chain (?s.Name = ?lp.subsystem)))
       (exists BIN       ?lb    suchthat no_chain (member   [?lp.bin ?lb]))
       (forall COMPILABLE ?C    suchthat
                 (or no_chain (member   [?lp.files ?C])
                     no_chain (member   [?mp.files ?C])))
       # verify that there is no local Main file
       (forall CFILE      ?lm    suchthat
                 (or (and no_chain (linkto   [?lp.main  ?lm])
                          no_chain (ancestor [?lp ?lm]))
                     (and no_chain (linkto   [?mp.main  ?lm])
                          no_chain (ancestor [?mp ?lm]))))
       # get master main file
       (forall CFILE      ?main  suchthat
                 (and no_chain (linkto [?s.main ?main])
                      no_chain (ancestor [?s ?main]))))
  :
 # condition
 (and  no_chain   (?lm.Name = "")          # Not-Exists condition
       no_forward (?C.compile_status = Compiled)
       no_chain   (?main.compile_status = Compiled))

 # activity
 # Use main object code with the SUBSYSTEM.object
  COMBINE_TOOLS build_local ?lb.executable ?lp.build_log
                ?s.libraries ?s.build_order
                ?s.med_libraries
                ?s.object_code ?C.object_code

 # success and failure effects
 (?lb.build_status = Built);
 (?lb.build_status = NotBuilt);
```

Figure 3.6: Treaty Process Steps for 3-site Builds

Chapter 4

Integrating a Standard OMS

Abstract

The integration of a legacy system and a standard Object Management System (OMS) is often a very challenging task. This chapter details a case study, our experiment interfacing between Oz and a PCTE (Portable Common Tool Environment) Object Management System. Oz is a multi-user process-centered software development environment that has been under development in our lab since 1987, originally under the name Marvel. PCTE is a specification that defines a language-independent interface providing support mechanisms for software engineering environments (SEE). One of the premises of PCTE is that, in theory, an SEE such as Oz can be built (or extended) using the services provided by PCTE. The purpose of our experiment was to study how a legacy system such as Oz can be integrated into a new environment framework, e.g., PCTE. The architecture of the legacy system and the services of the framework are the key factors in the integration approach. Because Oz historically has included a native OMS, our experiment focused on modifying Oz to use the PCTE OMS, which has an open and standard API. This chapter describes how several Oz components were changed to interface to the PCTE OMS. The resulting proof-of-concept hybrid system has process control and integration services provided by Oz, and data integration services provided in part by PCTE. We discuss in depth the solutions to the concurrency control problems that arise in such an environment, where Oz and PCTE use different approaches to transaction management (i.e., each has its own transaction manager). The PCTE implementation used in our experiment was the Emeraude PCTE V 12.5.1, and the Oz version was V 1.0.1.

4.1 Introduction

The goal of a software development environment (SDE) is to provide support for software engineering activities such as system design, implementation (coding), testing and documenting. Support is more effective if the environment is integrated – if all its components function as a single, consistent, coherent and integral unit [223]. There are several key aspects of integration: tool, data and process. Tool integration provides a development tool set and an invocation mechanism to control its use within an environment. Data integration normally is based on an object model of software artifacts: it uses object management technologies to handle the repository, duplication, sharing and consistency of the artifacts. Process integration normally uses a software process model to explicitly represent the software development activities and the workflow among activities; it guides and coordinates development activities and integrates tools and data in the environment. Process integration is at a higher level than tool and data integration [161]. In fact, the invocation order of tools to perform routine development tasks, e.g., compiler (to compile) then linker (to link), in an environment implicitly defines the conditions and orders of tools. Similarly, the production and consumption of software artifacts normally has a partial order during a software life cycle, e.g., executables are produced before being used in test runs. It is very clear that a process, manual or automated, is required to insure the proper order for tool invocation and software artifacts production and consumption. Therefore an important goal of many integrated software engineering environments is support for the definition, enforcement and automation of a software process.

PCTE is a specification of an environment framework which provides support for environment builders to write, assemble and customize integrated software development environments. A number of systems, for example, the Object-Oriented Tool Integration Services (OOTIS) [94] from IBM and the PCTE Workbench [68] from Vista Technologies, have been successfully built on top of PCTE. An SDE that has built-in support for process integration is called a process-centered environment (PCE). Oz is a process-centered, multi-user and multi-server environment. It (with its predecessor, Marvel) is one of the oldest PCEs still in existence. Regarding PCTE as a good example of a standard framework and Oz as an example of a legacy PCE (SDE), the main goal of our experiment was to make a transition from Oz to the PCTE framework. Ultimately, we wanted Oz to conform to the PCTE standard and be able to integrate with other software tools in a PCTE environment. We believe that such an experiment is valuable because:

First, we can test the capabilities of the PCTE framework in facilitating the migration of legacy SDEs. We feel that it is very important for a standard framework, such as PCTE, to be able to support legacy systems, such as Oz, because while adapting to a new standard the investments in the existing SDEs should be preserved. The architecture of a legacy system and the types of standard services and their APIs provided by the framework, are the key factors in determining the approaches to the transition. Oz [21] has several major components: the Process Engine, the Transaction Manager, the Object Management System, and the Activity Manager. They were loosely coupled through the access functions of each component. In particular, since the services provided by the Object Management System are conventional object storage and manipulation operations, we felt that we should be able to replace this component with another objectbase without much changes to other Oz components. The PCTE OMS is open and standard, its services and the API address many data integration problems. Thus, a natural pathway in moving Oz to PCTE is for Oz to use the data integration services provided by PCTE, in other words, for Oz to interface with the PCTE OMS.

Second, we can also study whether it is feasible for a standard framework and a legacy SDE to complement each other in providing support for integration. A commonly recognized limitation of PCTE is the lack for the sort of high-level control integration capabilities (it only has low-level controls similar to the basic Unix mechanisms) on which much of process integration support is built [152]. It is typically the case that given any single framework there is strong support for some forms of integration but only weak support for other forms. Thus to remedy weak support for a certain form of integration, it is often necessary to add additional support - in our case from another environment (or framework). Since Oz has strong support for process integration, we wanted to determine whether Oz can provide process integration support in a PCTE environment.

The challenge of moving a legacy SDE into a standard framework (vs. developing a new SDE using the framework) lies in the potentially substantial changes required in the SDE (since we can not assume that the standard framework can be changed to suit the legacy system - in fact, this would defeat the purpose of the "standard"). The exercise of making the changes can reveal the adaptability of the legacy SDE as well as the limitations of the framework. As a major side effect, our experiment tested the functionalities of the PCTE OMS and the ease of replacing an Oz component, its native OMS, with an outside component, the PCTE OMS.

In a truly componentized system, the OMS component can be replaced by another objectbase management system without changing the internals of other components, since ideally the OMS provides an API from which other components access the objectbase. Thus ideally only a translation layer that maps the Oz OMS API to the PCTE OMS API would need to be implemented for Oz to use the PCTE OMS. However, the implementation of Oz had incorporated a large amount of legacy code from Marvel, the predecessor of Oz. As a result, the native OMS had a weak and incomplete API and other modules made fixed syntactic format (e.g., using pointer operations to access objects) and semantic assumptions (e.g., files are stored in the host file system rather than as first class objects in the objectbase) on the underlying data model and data manipulation primitives. Since the PCTE OMS is different from the Oz native OMS in both the data model and the set of primitives, we needed to deal with these fixed syntactic and semantic assumptions. We made changes mainly to two sets of Oz functions in our experiment. The first set of functions was the object storage and retrieval functions, which were changed to use the PCTE OMS API [64]. We tried to preserve the internal (run-time) format of Oz objects by reading PCTE objects into the Oz data structures so that code that used the fixed syntax (e.g., pointers) did not need to be changed. The second set of functions was the concurrency control functions, which were expanded to incorporate PCTE activities [84] (sequences of object operations) into Oz transactions.

This chapter is organized as follows: Section 4.2 describes briefly the services provided by PCTE, with the PCTE OMS covered in detail. Section 4.3 provides an overview of the functionalities of Oz components. Section 4.4 compares the Oz OMS model and the PCTE OMS model, and discusses how to define an Oz data model using the PCTE OMS. Section 4.5 describes the implementation of the Oz interface to the PCTE OMS. This interface consists of object operations implemented using the PCTE OMS primitives. Section 4.6 examines the concurrency control problems that arise when the PCTE OMS is in use. Two different solutions are discussed in detail. Section 4.7 describes an example Oz environment that uses the PCTE OMS as the data repository. Section 4.8 compares our experiment to other related work. Section 4.9 concludes this chapter with a summary of our experience and a discussion of possible future work.

4.2 PCTE Overview

PCTE, for Portable Common Tool Environment, is a framework for CASE tool integration. Tool integration not only means that all of the tools exhibit some measure of uniformity, but also that they should be able to share the same data and communicate with each other. The commonly accepted solution for the development of environments that facilitate CASE tool integration is to use an Open Software Integration Platform, known as an Environment Framework. PCTE is a widely supported framework based on the Reference Model for Software Engineering Frameworks developed by ECMA and NIST [168]. This model suggests that a framework should support at least three main sets of services, namely, User Interface Services, Communications Services and Object Management Services, which correspond to these three dimensions of tool integration: presentation, control, and data. The fourth important set of services of the framework, the Process Management Services, is to support the development of an environment enforcing a particular software development process. Note that a process should not be "built-into" a software development environment. Instead, the environment should provide facilities for users to define and tailor a particular process suitable for the particular software development policies and activities.

For presentation integration, PCTE supports Motif-compliant tools. In the area of data integration, PCTE provides a single, common data repository, open to all components of the environment. This data repository provides all the basic services necessary in an Object Management System (as described in the ECMA/NIST Reference Model) for a high level of data integration between tools. For control integration, PCTE provides message and notification facilities.

The Emeraude PCTE environment is a distributed SDE. It typically consists of a network of hosts, which are workstations that allow users to access the Emeraude PCTE environment. It provides an operating system-like environment where users can log into the PCTE shell and execute PCTE commands (e.g., objectbase navigation commands) or invoke integrated tools. It also provides APIs (in various programming language bindings) that implement the PCTE specifications. These APIs enable developers to implement or re-engineer software tools to integrate them into the PCTE environment.

4.2.1 The PCTE OMS Model

The main focus of our case study is on the OMS services provided by PCTE. These services include data typing and data storage along with concurrency control. This subsection briefly explains the key concepts and features of the PCTE OMS model.

The OMS model defines the static information about the data stored in the objectbase, which is the repository of all persistent PCTE data (both the product and control data of an SDE). The objectbase is managed by the PCTE OMS, and (in the Emeraude PCTE environment) is distributed over the environment on a number of logical volumes (partitions of of the objectbase) corresponding to the physical distribution of storage devices over the network. A volume is mounted on a Unix file system or a device of a particular host so that the data held in the volume is available transparently throughout the environment.

4.2.1.1 Object Model

The objectbase represents discrete items of data as typed objects connected in a logical network by typed links (and relationships). Objects and links have typed attributes.

Objects represent the basic entities upon which operations are to be performed, and one object exists for every distinct entity that is to be operated on by a tool. Examples of objects are text files, documents, source and compiled modules.

Links represent relations between objects. A link is directional, emanating from one source object to the other destination object; it is always paired with a reverse link going in the opposite direction, either as part of a **relationship** (paired mutually dependent links) or created automatically as a system reverse link. An example is a link *writes* from object *user* to object *document*, representing the fact that *document* is created by *user*. The link in this example is called a **reference link**, which models dependencies between objects. Another type of link is the **composition link**, which models composite objects. A composition link is automatically created by PCTE when an object is created. It links an existing object (the parent object) to the newly created object (the child object). An example is the composition link *contains* from object *book* to *chapter_1*. A link also has a **cardinality**, either one or many. This defines whether more than one link of this link type can start from the same source object.

Direct reference to a particular object or link in the objectbase is through navigation. A pathname that walks the composition hierarchy top-down can be used to access an object or a link. For example, assuming *book* is a top-level object, then the pathname to access object *chapter_1* of *book* is *book/chapter_1*. Which objects and links can be seen and created at any given time depends on the **working schema** of the currently running PCTE process (on behalf of a tool). A working schema contains all the type information needed by the tools being executed. The use of working schema allows different views of the objectbase to be presented to different tools so that only the appropriate types are visible to a particular tool.

4.2.2 Communicating with the PCTE OMS

Tools communicate with the OMS through a number of tool interfaces provided by PCTE. These interfaces include the C language interface, which is a set of C libraries that can be linked with the object code of tools programmed in C; and the PCTE shell interface, which allows users to communicate with the OMS directly through a set of OMS commands.

Tools that are integrated into or developed within a PCTE environment are stored in the PCTE objectbase as objects (of type *Sctx*) and their executions (at run-time) are managed as PCTE "client" processes. These processes are not stored objects because they are purely dynamic. PCTE provides facilities that are similar to Unix to manage the start, stop, suspension, communication and synchronization of PCTE processes [221]. On the other hand, Unix tools that are interfaced to a PCTE environment need to be linked with the PCTE Unix libraries, and their executions (which are standard Unix processes) are "alien" processes to PCTE. In our case study, we modified Oz and linked it with the PCTE libraries instead of integrating it into PCTE.

4.3 Oz Overview

Oz is a process-centered software development environment that supports cooperation among engineers who follow a project-specific software development process (workflow) [21]. It facilitates both data sharing and process coordination. The architecture of an Oz environment is based on the client/server model. An Oz server manages the objectbase that stores both product and control data of a software development project. It also automates and enforces the project-specific process. An Oz server can support multiple Oz clients sharing the same process and accessing the same objectbase. A user interfaces directly with an Oz client, which in turn communicates with an Oz server. An Oz client provides a user interface from which a user can view the objectbase hierarchy, query the objectbase, and can perform development tasks (process steps), e.g., edit, compile, and build. In general, a task (process step) execution cycle in Oz is the following:

- The Oz client sends a task execution request (from the user) to the Oz server.
- The Oz Server checks the prerequisites of the task according to the process definition, and sends the necessary data (e.g., a C file object) back to the client to perform the task. In case that the prerequisites are not satisfied, Oz server may invoke other process steps whose results may satisfy the prerequisites of the current process step.
- The Oz client then invokes the appropriate tool (e.g., an editor) on the product data.
- After the user finishes the task (exits the tool), the Oz client sends back the data (e.g., a changed C file object) to the Oz server.
- The Oz server propagates appropriate changes to the objectbase, and according to the process definition, may invoke other process steps if their prerequisites are now satisfied because of the changes made by the current process step.

4.3.1 Major Oz Components

As stated earlier, the architecture of Oz is one of the most important factors in our approach of moving Oz to PCTE. We now briefly describes the major components of the Oz:

The Process Engine. Tasks are the key elements of a process definition. Each task definition consists of a name; a list of typed parameters; a condition consisting of bindings of local variables and a complex property clause that must hold on the actual parameters (instances of control and product data) and bound variables for the task to take place; an optional (physical) activity that specifies a tool envelope (see Section 4.3.3) and its arguments; and a set of mutually exclusive effects, each consisting of assertions to the objectbase that reflect one of the possible results of executing the activity. Tasks are implicitly related to each other through matches between a predicate in the condition of one task and an assertion in the effect of another task. Process enactment is done through the dynamic construction of task segment. Backward chaining of a sequence of tasks is attempted to satisfy the condition of a task. Forward chaining through "atomicity chains" is performed to propagate data changes to preserve the atomic nature of a transaction. Forward chaining through "automation chains" is for the purpose of automating sequences of activities, not necessary atomically.

The Transaction Manager is a separate component called Pern [102], which interfaces to the rest of Oz (or another environment framework) through application-specific mediator code. Because multiple clients can access the same objectbase, conflicts may arise among user activities and concurrency control is thus in order. Oz follows the approach of associating tasks with transactions. A task segment consists of all tasks executed during backward chaining, followed by the user-invoked task (which caused the backward chain), followed by all tasks executed during forward chaining. Pern supports a nested transaction model in which a task segment corresponds to a series of top-level transactions. Each atomicity chain is a subtransaction of the triggering transaction and each task in an automation chain starts an independent top-level transaction.

The Object Management System (OMS). In an Oz environment, all data, both process control data and product data, are stored in and managed by the OMS. The Oz OMS contains an Object Manager, a Storage Manager, and a File Manager. The Object Manager implements the data model, provides persistence, and performs all requests for access and modification of both control and product data. It assumes an object-oriented data model, with a class inheritance lattice, an object composition hierarchy and arbitrary relationships between objects in the same objectbase. The Storage Manager is responsible for low-level disk and buffer management for control data. Since Oz integrates file-based external tools and maintains its product data in ordinary files, the File Manager is responsible for accessing files requested by other Oz components. Usually, the data model of a process encapsulates the product data within control data and abstracts the file system as objects by providing typing and relationship information. In this case, the File Manager is just a mapping function between objects' file attributes and their file contents. The Oz OMS also provides an ad hoc query processor and a set of object manipulation functions.

4.3.2 Communicating with the Oz OMS

All the major Oz components need to communicate with the OMS. The Process Engine queries the objectbase to obtain the actual parameters of a task and evaluates its condition. The task's activity accesses and modifies data in the objectbase. The Process Engine also modifies data in the objectbase when it asserts an effect of a task. Pern locks objects accessed within a transaction and directs the OMS to undo changes in the objectbase when a transaction is aborted.

At the time of this experiment, the Oz components were tightly coupled with the OMS. For example, it was assumed that the whole objectbase is loaded into Oz buffers when the Oz server is started and access (read) to objects is then done by directly addressing (using memory pointers) into these buffers instead of function calls to the OMS interface. Object modification (updates) is done by a set of OMS functions that handle both updating the in-memory buffer and the on-disk persistent data. But these functions only work with an assumed data model where objects can have *file* attributes, which are files stored in the hidden file system, and *composite* attributes, which refer to one or more component objects.

4.3.3 Oz Tool Envelopes

A typical Oz process may utilize several external tools to carry out software development activities. Oz provides one or more tool envelopes for each external tool so that the external tool can be used as a "black box". In our case study, we developed a special envelope to wrap around some PCTE related object operations, see section 4.6.3. A tool envelope represents the implementation of an activity and is executed by Oz client. Tool envelopes are written in SEL, an extended Unix shell language [86]. SEL allows the tool integrator to write shell-like code to wrap around the call to the tool. It also requires the explicit declaration of all object attributes, along with the corresponding types, that are input or output variables of an envelope. This requirement hides the details of the object data model from the envelope writer. SEL also handles multiple output (return) values so that it can not only report the status of tool execution back to Oz, but also return the execution results, which may be assigned to object attributes in a task's effects. A tool envelope written in SEL is translated using the Oz *make_envelope* utility into a standard Unix shell script (currently, *sh*, *csh* or *ksh*).

4.4 Implementation – Data Model Mapping

In an Oz and PCTE OMS hybrid system, Oz objects (all control and product data of an Oz environment) are to be stored in the PCTE objectbase. Thus we need a general solution (not ad hoc for each Oz environment) of using the PCTE object model to define the data schema used by an Oz environment. Since an Oz data schema is defined according to the Oz object model, we need to first compare the Oz object model and the PCTE object model, then devise a mapping between the two.

4.4.1 Comparison of Oz and PCTE OMS Models

4.4.1.1 Similarities

Since both the Oz object model and the PCTE object model are intended to be general-purpose, there are many similarities between the two.

- Both Oz and PCTE support the concept of an object, that can be uniquely identified in the objectbase – in Oz via the unique object identifier (OID) and in PCTE via the navigational pathname to the object.
- Both associate a type (class) with an object. Both support a type hierarchy and inheritance. However, PCTE only allows single inheritance while Oz supports multiple inheritance.
- Both have a pre-defined root in the type hierarchy – in Oz the *ENTITY* class; in PCTE the *object* type.
- Both apply attributes to objects. In Oz, primitive attributes can be of string, integer, boolean, date (time stamp), real, and user-defined enumerated types. In PCTE, only string, integer, boolean, and date are allowed.
- Both use links to model binary relations between objects. Both have implicit reverse links to facilitate objectbase navigation.
- Both support an object hierarchy through composite objects. The definition (and construction) of the object hierarchy is identical. In Oz, an object can have *composite* attributes each of which can be a component object or a set of component objects. In PCTE, a parent object can have composition links to one or a set of component objects.

4.4.1.2 Differences

The main differences between the two object models are:

- In PCTE, type definitions (classes) are stored in the objectbase as first class objects; in Oz, class definitions are stored separately from the objectbase in a file and loaded into the Oz server at startup.
- In Oz, the name space of attributes and links is scoped within each class where the attributes and links are defined and applied. In PCTE, all objects, links and attributes in a schema share a single name space.

- Oz objects may have file attributes whose values are the pathnames of files in a "hidden" file system (with directory hierarchy and file names not comprehensible outside of the Oz system). It is therefore possible to associate an Oz object with one or many files. In PCTE, objects that can have file contents are of type, or subtype, of the *file* object type predefined in the PCTE OMS. A PCTE object can be associated with at most one file.

4.4.2 Using the PCTE Object Model to Define Oz Schemas

The following are general guidelines for defining an Oz data schema in the PCTE OMS:

- Each link and attribute name in PCTE is prefixed with the object type (class) name. For example, attribute *title* of class *MANUAL* is defined (named) as *MANUAL_title* (see Figure 4.2). This guarantees that links and attributes names are unique in a schema.

- Oz link attributes are modeled as relationships in PCTE and each composite attribute (for component object) in Oz is mapped to a composition link.

- A file attribute for a file in Oz is mapped to a composition link to an object of type *Oz_FILE*, which is a subtype of *file* in PCTE. For example, if an Oz (object) class *cfile* has a *file* attribute named *contents*, then in PCTE, *cfile* will have a composition link *Oz_FILE* named *contents*.

- *Oz_OID* and *Oz_NAME* are Oz system-attributes for all Oz objects in PCTE. Each *Oz_OID* has a unique value in an objectbase; it is added to facilitate Oz components in referencing objects. *Oz_NAME* is needed mainly for the purpose of objectbase display in Oz clients. *Oz_OID* and *Oz_NAME* can therefore be regarded as the external identifiers of an object, with the first used in the programming interface and the second used in the user interface. The pathname of an object can then be regarded as the internal (navigational) identifier.

- An attribute of a user-defined enumerated type is mapped to an attribute of type string in PCTE. While the default value can be set, the list of the possible valid (enumerated) values is not stored. Therefore we omitted the validation of enumerated attributes for the sake of simplicity.

We limited our case study to those Oz data models that do not use multiple inheritance in their class hierarchies. Although one can always devise an algorithm that translates a multiple inheritance class hierarchy into a single inheritance class hierarchy, it is neither the goal nor the interest of this experiment to actually implement such an algorithm. We have found few significant fixed assumptions about multiple inheritance in Oz components.

Suppose we have an Oz class definition as shown in figure 4.1, which defines the class *MANUAL* as having base attributes *title*, *reformat* and *format_status*, consisting a set of objects (*submanuals*) of class *SUBMANUAL* as its composite attributes (component objects), a file attribute (*first_page*), and a link (*first_sub*) to an object of class *SUBMANUAL*. Then the relevant information in the PCTE schema definition file will be as shown in figure 4.2. Here, we can see that the expansion factor (in terms of lines of definition) is about two because each attribute (base attribute, link, component) has to be first defined individually, and then be referenced in the class definition.

4.5 Implementation – Interface to the PCTE OMS

Figure 4.3 depicts the simplified (run-time) system architecture of Oz interfacing with the PCTE OMS. The arrows represent data flows. Here we see that Oz modules make direct reference to (read from) the object buffers instead of always using the OMS API functions[1]. This means that Oz components assume that the entire objectbase is loaded into memory when the Oz server is started.

[1]this is no longer the case in Oz Version 1.2.1

```
MANUAL :: superclass ENTITY
 title : string;
 reformat : boolean = false;
 /* enumerated-type attribute */
 format_status :
  (Initialized, Working, Done) = Initialized;
 /* composite attribute */
 submanuals : set of SUBMANUAL;
 first_page: text /* file attribute */
 first_sub : link SUBMANUAL;
```

Figure 4.1: An Oz Class Definition

While this assumption could potentially be changed, we were mostly interested in investigating how the process engine and transaction manager need to be changed when an external OMS is used. We wanted to minimize our efforts while achieving our goals. Therefore we tolerated this fixed assumption by loading the PCTE objectbase into the in-memory buffers in the Oz server. The Oz *read_objectbase* function was rewritten to read the PCTE objectbase.

Once the objectbase is loaded into Oz buffers, objectbase navigation can be done in memory using the existing Oz functions without changes. On the other hand, the effects of object update operations are written immediately to the PCTE objectbase. This is done to facilitate data sharing with other tools running in the PCTE environment. The PCTE-specific code (that implement the updates) is isolated to a handful of low level functions because there is already some degree of abstraction in Oz – the components do not have knowledge of how the objects are physically stored: low level functions (in the Storage Manager) handle the details. The PCTE objectbase has effectively replaced the storage manager.

4.5.1 OMS Primitives in PCTE

Before we go into the details of reading and updating the PCTE objectbase, we first briefly describe the APIs of the PCTE OMS.

The PCTE OMS C language binding contains a rich set of OMS primitives, which are essential for developing an access interface for an external system (such as the Oz server) to communicate with the PCTE OMS:

- Data Definition Primitives: schema definition and browsing.
- Data Manipulation Primitives: object, link, and attribute manipulation.
- Establishing the Working Schema – establish a working schema consisting of a set of schema definitions. Tools can access only the objects, links, and attributes defined in the working schema.

4.5.2 Reading the Class and Object Hierarchy into the Oz Server

When the Oz server starts up, it first reads the class hierarchy of the data model in use and then reads the whole objectbase. We now describe our re-implementation of these two functions.

Every Oz environment is associated with a file system directory, called the environment directory, which contains environment-specific process definition files, tool envelope files, an objectbase file, and a subdirectory containing the "hidden file system", which stores the file attributes of objects. The *strategy* file contains the class hierarchy in an internal format, which is loaded into the Oz server when it starts. Thus Oz effectively employs a single (working) schema for each Oz server.

```
OZ_OID : string;
OZ_NAME : string;
MANUAL__title : string;
MANUAL__reformat : boolean := false;
MANUAL__format_status : string
 := ''Initialized'';
/* OZ_OID is the key attribute of this
 * cardinality many link
 */
MANUAL__submanuals : composition link
 (OZ_OID) to SUBMANUAL;
/* file attribute is mapped to a composition
 * link to a file object
 */
MANUAL__first_page: composition link
 to Oz_FILE;

/* link mapped to relationship */
relationship (
 first_sub : reference link to SUBMANUAL;
 first_sub_of : implicit link to MANUAL
);

ENTITY : subtype of object;
MANUAL : subtype of ENTITY;
Oz_FILE: subtype of file;

extend ENTITY
 with
 attribute OZ_OID;
           OZ_NAME;
end ENTITY;

extend Oz_FILE
 with
 attribute OZ_OID;
           OZ_NAME;
end Oz_FILE;

extend MANUAL
 with
 attribute MANUAL__title;
           MANUAL__reformat;
           MANUAL__format_status;
 link MANUAL__submanuals;
      MANUAL__first_page;
end MANUAL;
```

Figure 4.2: Corresponding PCTE Schema Definition

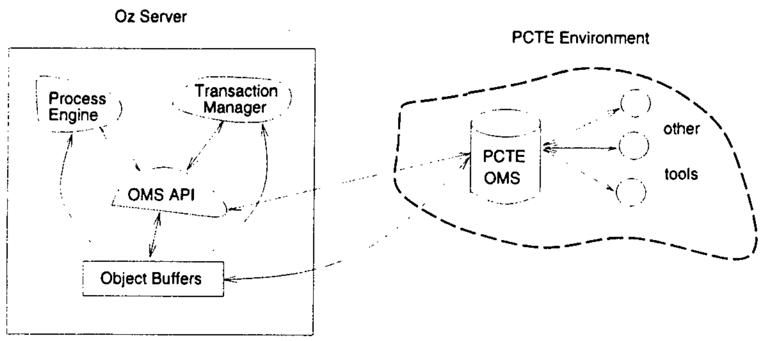


Figure 4.3: System Architecture of Oz Interfacing with PCTE OMS

In PCTE classes are instead first class objects that are actually stored in the objectbase, and type definitions are visible within a specific working schema. We put the keyword *PCTE*, followed by the names of the schema definitions that constitute the working schema, in the Oz *strategy* file. This information is used by the Oz server to first set the working schema. Data definition browsing primitives (of the PCTE OMS) are then used to get all the type definitions in the PCTE objectbase. For each object type definition, its parent type definition is used to build the class hierarchy. Also for each class, the definitions of its attributes of type integer, string, boolean or date are read. The definitions of its composition links are mapped to Oz composite attributes definitions; the definitions of its relationships are mapped to Oz link attributes definitions (recall from section 4.4.2 that we use PCTE relationships to represent Oz links and PCTE composite links to represent Oz composite attributes).

Reading the objectbase from PCTE is achieved through navigation starting from a set of top-level objects. A list of absolute pathnames of PCTE top-level objects is thus the only needed contents of the *objectbase* file in the Oz environment directory. To read the object hierarchy from these top-level objects, depth first search (DFS) or bread first search (BFS) can be used.

We use a modified DFS in our implementation. The modification is the following: when an object A is read, if A has a reference link (as part of a relationship) to object B, then B is read (recursively using DFS) if B has not been read already. This is done to facilitate the set up of the reference link and the reverse link between A and B. If we used the standard DFS, B would not be read at this point since B is not a child (a component object) of A. The consequence would be that the reference links of A can not be set up when A is being processed, thus a second pass would be needed to set up the reference links between objects. It is thus clear that our modified DFS is more efficient.

4.5.3 Oz Built-in Operations

There are a number of built-in object operations in Oz, namely *add*, *delete*, *copy*, *move*, *link*, *unlink*, and *rename*. These operations first update the in-memory copy of the objectbase and then write the updates to the persistence storage. The functions to update the objectbase had to be rewritten using PCTE OMS primitives. These changes were in general very straightforward.

The exception was the *move* operation, which moves object *from* to object *to* so that *to* has an exact copy of *from* as its child object and the original parent object of *from* no longer has it as a child object. PCTE only provides primitives to move an object across physical storage volumes (in PCTE, the objectbase can be partitioned into multiple volumes). Since we do not assume *to* and *from* are in different physical volumes, the *move* operation has to be done in a number of steps. In the first step, the whole *from* object hierarchy is copied and placed under *to*. The links are not copied; instead, for each link, the object ids of the source object and the destination object, along with the link name, are stored in a table called *link_table*, which has entries of the form $(from\_id, to\_id, link\_name)$. For each object being copied in this step, ids of the original object and the copied object are stored as (*old_id*, *new_id*) in table *object_table*. In the second step, the original *from* object hierarchy is deleted. The links are also deleted as a result. In the third step, *object_table* and *link_table* are used to create links. For example, a link recorded as (A, B, l) is now created as link l from A' to B if (A, A') is found in *object_table*.

4.5.4 Accessing PCTE Objects in an Oz Task

As discussed in earlier sections, the Oz process engine needs to access objects during all phases of process enactment, namely, bindings of task inputs, checking of prerequisites, execution of task activity, and assertion of task effects. Since the Process Engine only calls the high level built-in operations to modify the objectbase, there should not be any changes required after the built-in functions have been changed to interface the PCTE OMS. However, in our experiment, we have found some fixed semantic assumptions about how files are modeled, that are beyond the scope of the built-in object operations and need to be dealt with separately.

In the Oz OMS, a file contained within an object is represented by one of its *file* attributes, while in PCTE it is a component object of subtype *file*. When reading the class hierarchy and objectbase from PCTE into Oz, file objects are treated the same as other component objects, i.e., they are translated into Oz *composite* attributes of their parent objects. As a result, only the process engine code that handles *file* attributes had to be changed to also consider *composite* attributes. The attribute type definition, in this case class *Oz_FILE*, is used to determine whether the *composite* attribute is a file object.

A more serious problem is when a task is applied to an object that has a file attribute in its object type definition but has no such physical attribute in the object instance yet. In the Oz OMS, the file attributes stored in the objectbase are just file paths to the host file system, which are known and fixed once the object instance is created. In the original Oz implementation, when an object is read into the Oz server (at server startup time) or when it is created in Oz, a file attribute pointer is created for each of its file attributes, regardless of whether the file exists or not. This works because the file path for a file attribute is already known even if the file is not physically created yet, so long as the object instance exists. When a task activity that requires access to a file attribute is executed, Oz just passes the file attribute (the file path) to the task; it then relies on the tool envelope (e.g., invocation of the editor on the file path) to create/modify the file.

For example, suppose we have class *C_FILE* with file attribute *C_contents*; an instance of *C_FILE*, *foo.c*, can be created without having physical *C_contents* due to, for example, that *foo.c* has never been edited. When a task, say *edit*, is applied to *foo.c* the first time, it requires access to its *C_contents*. Oz will pass to the editor the *C_contents* attribute, which is just a file name (for a nonexistent file) in the host file system. After *edit* completes, the *C_contents* is now the (unchanged) file path to an existing file. When the PCTE objectbase is in use, however, files are stored as first class objects in the objectbase, and a file attribute of an object is modeled as a component object of object type *file*. Therefore, creating a pointer (in memory) to a nonexistent component object is obviously not acceptable because it will seriously compromise the data integrity of the objectbase in memory. Our solution to this problem is the following: the file component object is created only when a task that requires a file attribute parameter is applied to this object. This action of creating the component is logged automatically by the governing transaction; therefore it can be undone in the case of a transaction abort (see next section). Since most external tools that access files require that files be in the host file system instead of in the objectbase, file copying in both directions is also required. Details are discussed in section 4.6.3

4.6 Concurrency Control and Recovery

Since Oz is a multi-user system with multiple Oz clients sharing the same objectbase, concurrency control is needed to prevent chaos. For the same reason, PCTE also provides concurrency control mechanisms for concurrent foreign and PCTE tools to cooperate with each other. Note the Oz server

compatible modes	read-unprotected	write-unprotected	read-protected	write-protected	write-transaction
read-unprotected	*yes*	*yes*	*yes*	*yes*	*yes*
write-unprotected	*yes*	*yes*	*no*	*no*	*no*
read-protected	*yes*	*no*	*yes*	*no*	*no*
write-protected	*yes*	*no*	*no*	*no*	*no*
write-transaction	*yes*	*no*	*no*	*no*	*no*

Table 4.1: Compatibilities of PCTE Lock Modes

may be only one of these (foreign) tools, and multiple foreign and PCTE tools may be accessing the PCTE objectbase at the same time as Oz.

Pern, the transaction manager used by Oz, is a flexible, powerful and elaborate system. In contrast, PCTE provides only the basic primitives for concurrency control. In this experiment, we try to make Pern and the PCTE OMS work together in such a way that Pern manages the high (task) level transactions and PCTE handles the lower (object operations) level subtransactions. Data locking is done first in Pern (in memory) and then in PCTE (in the objectbase). If a lock can not be acquired in PCTE, the current Pern transaction will normally abort.

There are certainly other more desirable approaches that we wish we could have used to integrate Pern and PCTE. However, as the following detailed discussion will show, the current implementation of PCTE has some serious limitations that make other alternatives very difficult to implement.

4.6.1 Concurrency and Integrity Control in PCTE

Concurrency and integrity controls in PCTE are enforced by means of **activities** (not to be confused with the activity of an Oz task). An activity in PCTE is a defined framework within which a set of related operations takes place. These operations involve accesses to the objectbase. Each operation is always carried out on behalf of a single activity.

An activity is characterized by its class. There are three classes of activity in PCTE:

- An **unprotected activity** does not require that its accesses to the objectbase be protected from other concurrent activities.
- A **protected activity** requires that its accesses to the objectbase be protected from the effects of other concurrent activities. The data it reads remain stable and the data it writes is not overwritten by other concurrent activities. Changes to the objectbase made by the activity are permanent even if the activity terminates prematurely.
- A **transaction** requires the same protection as a protected activity, but in addition it must be atomic; that is, all or none of its results should be applied to the objectbase. This corresponds to traditional database transactions, with full rollback of the objectbase updates when a transaction fails for any reason.

An activity can acquire **locks** on **resources**. Here a resource refers to: either an object with its contents and its set of attributes, but not the links originating from the object; or a link and its attributes. The effect of a lock on a resource is the following:

- **read-unprotected** mode allows other activities to write to this resource concurrently.
- **write-unprotected** mode allows other activities to write to this resource concurrently. Any modifications, from whatever source, are immediately applied.
- **read-protected** mode prevents other activities from writing to this resource concurrently. This mode is applicable to both protected activities and transactions.

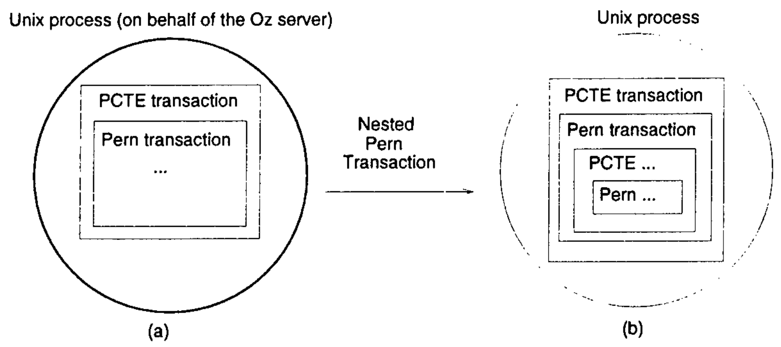


Figure 4.4: Nesting a Pern Transaction within a PCTE Transaction (not possible)

- **write-protected** mode prevents other activities from writing to this resource concurrently. Any modifications, by the activity which holds the lock only, are applied immediately.
- **write-transaction** mode prevents other activities from writing to this resource concurrently. Any modifications, by the activity which holds the lock only, are either applied or discarded once the lock is released. This mode is applicable to transactions only.

Compatibilities of lock modes are shown in table 4.1.

PCTE has a different implementation of nested transactions than many other DBMS's because of the association of process and activity. A process (an executing program), which carries out operations that access the objectbase, is always initiated in the context of an activity. A process can start and control new activities, which are treated as **nested** (internal) to the one in which the process was started. In the current implementation of PCTE in the Unix environment, the only way for a process to start a new activity is for this process to spawn a child process and then let this child process start an activity. Thus, within each Unix process, there can be only one (flat) PCTE activity.

In the following sections, we will see the consequences and limitation of this implementation. Basically, since in the Oz environment there is a single Oz server per objectbase and Pern is part of the Oz server, which is implemented as a single Unix process, there is only a single Unix process on behalf of Pern with respect to a given objectbase at any given moment. A Pern transaction corresponding to a task often requires nested transactions, for example, because of an atomicity task segment. If we try to put a Pern transaction within a PCTE transaction, then the implementation will require that a PCTE transaction is started before a Pern transaction is started, see part (a) of figure 4.4. If the Pern transaction happens to have nested subtransactions, a PCTE transaction should also be started before each Pern subtransaction is started. But this second PCTE transaction would then be nested within the first PCTE transaction, as seen in part (b) of figure 4.4, and they are both started by the same Unix process for the Pern (actually the Oz server) executable. This is not allowed by the current implementation of PCTE. Therefore there is currently no simple and direct way for Pern to take full advantage of the PCTE nested transactions facility. Instead, we have devised mechanisms that enable low-level PCTE object operations to be controlled by PCTE protected activities and transactions, while high level object manipulations (as in an Oz task segment) to be guarded by Oz's Pern.

4.6.2 Built-in Operations as PCTE Protected Activities

The built-in object operations in Oz, namely *add*, *delete*, *copy*, *move*, *link*, *unlink*, and *rename*, require write access to the PCTE objectbase and thus require protection from other concurrent PCTE activities. The semantics of these operations are well-defined and have no dependencies on other operations. Therefore they can be implemented either as protected activities or as transactions because either can provide the same protection against other concurrent activities. We decided to implement these operations as PCTE protected activities in order to gain more experience with PCTE, since we implemented some other object operations (discussed in the next section) as PCTE transactions.

Each of the built-in operations eventually calls a low level function that uses PCTE OMS primitives to carry out the object operations in the PCTE objectbase. These low level functions can be placed within PCTE protected activities. For example, *do_add_operation* calls the function *OMS_write_object* to write objects in the PCTE objectbase. Before calling *OMS_write_object*, a PCTE protected activity is started so that all object operations following this call are protected. When *OMS_write_object* returns, this activity is ended. Within *OMS_write_object*, before writing an object (changing its attributes or creating a child object), an attempt is made to lock this object in write-protected mode. If the lock is obtained, then this object is written to PCTE's OMS and the lock is released. This sequence of actions ensures that no other concurrent PCTE activities can write to this object if an Oz built-in operation is accessing it.

These built-in object operations are normally called upon by a task's effects and are therefore part of a Pern transaction. If the PCTE protected activities on behalf of these Oz built-in operations have to be aborted, for example, if an exclusive lock can not be obtained, the upper level Pern transaction also aborts. Say, a task's effect has two update operations: $add(a, b)$ and $link(c, d)$. These two operations both have to succeed. Suppose that the *add* succeeds but the *link* fails; Pern would then abort the transaction of this task. This requires the result of the *add* to be undone in the PCTE objectbase. Pern maintains an entry for each invocation of such operations in its log file so that the effects of these operations can be undone when a Pern transaction aborts. The details of putting entries in the log file and performing recovery for these PCTE object operations are discussed in section 4.6.4.

4.6.3 External Tools Wrapped in PCTE Transactions

The activity part of an Oz task can invoke an external tool such as the default editor. These external tools normally require access to the contents of objects (for example, the content of a C source file). Object contents, i.e. files, are of utmost importance in a (coarse-grained) software development environment and thus demand vigorous protection. For this reason we decided to wrap the tools that access PCTE object contents in PCTE transactions. This implementation ensures that the updates of object contents are all-or-nothing, and no intermediate changes can be seen by other concurrent PCTE activities (this is a given, since in our Oz/PCTE integration, a PCTE file object is always copied to the host file system before an external tool can use it). Recall the PCTE limitation that a Unix process can only have one PCTE transaction. If it did not have this limitation, we could easily implement the top level Pern transaction as a PCTE transaction and the tool invocation as a nested PCTE transaction.

We overcame this limitation by implementing a wrapper Unix program, a **transaction envelope**, that runs as a separate Unix process. This transaction envelope receives the message regarding what tool operates on what objects, along with some Oz environment-specific information (e.g., the id of the Oz client that runs the task activity) from the Oz client. It then starts a PCTE transaction, tries to lock the required objects, and then calls a **tool envelope**. Figure 4.5 (the arrows represent control and data flows) shows how the transaction envelope interacts with the Oz client and the PCTE OMS. It also shows that the PCTE transaction carried out by the transaction envelope is part of a Pern transaction. The tool envelope is a shell script that invokes the actual tool in the system, and reports the status and returns the result values to its caller. Normally the return code from a tool envelope is used to choose one of the task's several effects. We have designated a return code to indicate an aborted request. The transaction envelope then decides whether to commit or to abort this PCTE transaction according to the return status from the tool envelope. The return status of the transaction envelope is the same as that of the tool envelope so that in the case of a PCTE abort, the abort request will be propagated to the Oz server and Pern; otherwise, this return code is used to choose the effect as usual.

When an external tool is called as a task's activity, the Process Engine of Oz marshals arguments

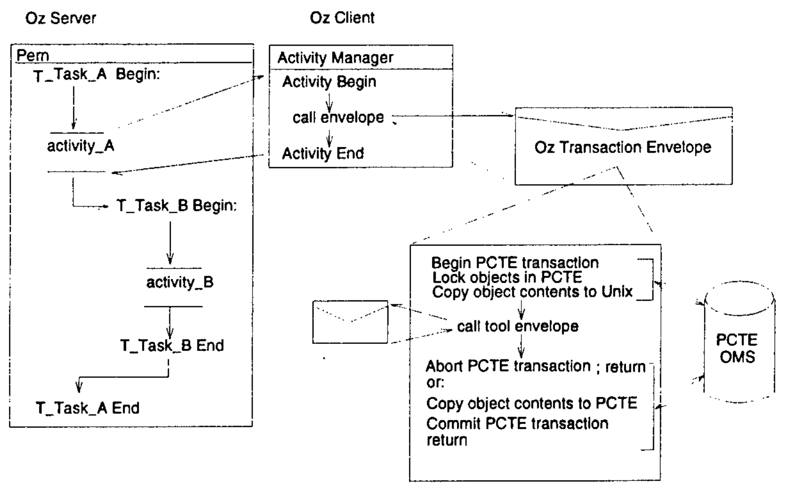


Figure 4.5: Transaction Envelope

(objects or attributes) required by this tool in a command line arguments buffer and sends it to an external program, the tool envelope, normally via the Oz client. The Oz client also creates a named pipe, a file in the host file system, and sends the pipe name to the envelope as a command line argument. This named pipe is used by the envelope to write the return values so that the Oz client can read them after the envelope terminates. When a transaction envelope is called, however, the command line arguments are marshaled differently than for a tool envelope. First, the name of the tool envelope (which the transaction envelope will call) is passed in as an argument; second, for each external tool argument that is of a file type, the Process Engine translates it to "*lock_mode* $*$ *PCTE_file* $>$ *Unix_file* $>$ *backup_file*", where:

- *lock_mode* is the kind of lock required on this object.
- *PCTE_file* is the full path of the object contents in the PCTE objectbase.
- *Unix_file* is the file (in the Unix file system) that initially contains the PCTE object contents. It is the file that the Unix tool (called by the tool envelope) will operate on.
- *backup_file* is the Unix file that maintains the original object contents.

The *lock_mode* indicates the lock mode that Pern requires on this object. For our experiment, it is S for shared or X for exclusive. At the point when the transaction envelope receives this message from the Oz server, the in-memory copy (within the Oz server) of the object has already been locked in an appropriate lock mode by Pern. The transaction envelope then tries to lock the persistent copy of the object in PCTE accordingly: X as write-transaction and S as read-protected. If a lock mode other than shared and exclusive is used in Oz, it also needs to be mapped to one of these two PCTE lock modes, which would generally require the "promotion" of the Oz lock mode. For example, if a SX (for shared exclusive) was requested in Pern, it would be promoted to X, which would then lead to a write-transaction lock in PCTE.

The object contents in PCTE, specified by *PCTE_file*, are copied to *Unix_file* before the tool envelope and thus an external tool (a Unix tool such as Emacs) is invoked. After the tool envelope execution, the contents of the *Unix_file* are copied back to *PCTE_file* before the transaction is committed. The *backup_file* is for the purpose of recovery; it is a Unix file that contains a copy of the object contents prior to the tool execution. The reason why we need this backup file is the following: although the tool execution is performed within a PCTE transaction, it is also part of a higher level Pern transaction, which may abort. But when this occurs, the changes to the object contents in PCTE have been committed and need to be undone. Our solution here is to put an entry in Pern's log file to indicate that the object contents have been updated and there is a backup file that contains the original contents. In the event of a Pern abort, the recovery function will use this log entry to copy the original contents back to PCTE, thus undoing the committed changes in PCTE.

Since the transaction envelope calls the tool envelope to actually execute the external tool, it creates a named pipe and passes the pipe name to the tool envelope so that the tool envelope can write the return status and output values to the pipe. The transaction envelope can then read the output after the external tool envelope terminates. After a PCTE commit or abort based on the return status, the transaction envelope just writes the same output to the named pipe that was passed from the Oz client, thus enabling the client to get the return status and output values. When the transaction envelope tries to lock an object in the PCTE objectbase, it may encounter a lock conflict because another PCTE tool is already holding the lock. In such a case, the transaction envelope aborts its PCTE transaction, then the upper level Pern transaction also aborts. The recovery is also handled by Pern.

Because file (PCTE object) copying is required for most of the Unix tool invocations, the performance of task activities when the PCTE OMS is in use must be worse than the case when the native Oz OMS is in use. Section 4.7.3 gives an example in tool performance comparison.

4.6.4 Recovery Using Pern Log and PCTE Transactions

In Oz, the Pern log file is used for the purpose of recovery. Each object operation has an entry in the log file. When recovery is needed, the entries are used to undo the effects of these object operations. Although Oz built-in operations are implemented as PCTE protected-activities and external tool invocations as PCTE transactions, they are both subtransactions of the upper level Pern transaction. Therefore their results need to be undone if the Pern transaction aborts.

To prevent a partial undo, we implemented each undo operation in a PCTE transaction. This ensures that undo can be done idempotently in case of failures during a previous undo.

4.6.5 Dealing with Concurrent PCTE Tools

Notice that the order of data lockings and the fact that PCTE transactions are subtransactions of Pern transactions only guarantees serializability between Oz activities within the same Oz environment, but not between an Oz activity and a PCTE activity that accesses the same PCTE OMS through its own interface. To see this, suppose that we have two Oz clients *A* and *B* that connect to the same Oz server, and they are firing tasks *task_A* and *task_B*, respectively. Further, the activities (or effects) of these two tasks try to gain exclusive access to a PCTE object *bar* at the same time. Since the execution of each Oz task is done within a Pern transaction, and there is a single Pern per Oz server, Pern can grant the exclusive lock on *bar* (locking done in memory) to either *task_A* or *task_B*. Actually, Oz (and hence, the Pern instance in use) requires that all the objects that a task intends to update be locked before the task's activity and effect take place. When, say, *task_A* gets the exclusive lock from Pern and actually accesses *bar* in the PCTE objectbase, it first does the actual locking of *bar* in the PCTE objectbase. This is done to prevent other external (outside the Oz environment) activities from accessing *bar* after *task_A* starts accessing *bar*. This mechanism (a strict two phase locking scheme) guarantees serializability between *task_A* and *task_B*, and is deadlock-free because: 1) *task_A* and *task_B* have to obtain all the locks at once from Pern; and 2) when *task_A* or *task_B* locks and accesses an object in the PCTE objectbase for which it has obtained the Pern lock, there can not be any other Oz task (within the same Oz server environment) that can access the same object.

The story is quite different if an external activity, say *C*, is accessing the PCTE objectbase through the PCTE OMS interface directly (without going through the same Oz server, and thus the same Pern). Unless *C* locks all the objects it intends to update at once before it actually writes to the PCTE objectbase, there is no guarantee of serializability between *C* and *A* (or *C* and *B*). For example, *C* can have an old copy of *bar* (done by reading without first locking *bar*) before *A* locks it,

```
task_A:          task_B:          external activity C:

read(X);         read(X,Y);       lock(X);
X=X-N;           Y=Y+X;           read(X);
write(X);        write(Y);        X=X+N;
                                  write(X);
                                  unlock(X);
```

Figure 4.6: Concurrent Activities

and then after A has committed the changes to *bar* and released the lock, C writes its own update of *bar*, resulting in a "lost update" problem (A's update is gone). The point here is that, unless all PCTE activities follow a transaction model such as two-phase locking in accessing the PCTE objectbase, there will be unpredictable results in the face of concurrent activities.

However, even if we are in a "perfect world" where all PCTE activities use the transaction model in accessing the PCTE objectbase, there can still be a "Incorrect Summary Problem" (also known as the "Inconsistent Analysis Problem") [60]. To see how this can happen, suppose that we have two tasks *task_A* and *task_B* and they are chained together (from *task_A* to *task_B* by an atomicity chain). The activities of *task_A* and *task_B* are shown in figure 4.6. The two tasks are chained because of, for example, a consistency constraint: "whenever X is updated, update Y accordingly". In Oz, *task_B* will be in a Pern subtransaction of *task_A*. X is locked in *task_A*'s transaction, and the lock is passed to *task_B*'s transaction. But when the transaction envelope is used, the activities of *task_A* and *task_B* will be in different PCTE transactions, and passing locks between the two envelopes, which are two different Unix processes, is not possible because a PCTE object can be locked by only one activity (in this case, on behalf of a transaction envelope, a Unix process) in any single moment, and the lock is released when the activity (and hence the PCTE transaction) is ended.

Imagine that after the transaction envelope for *task_A* commits the corresponding PCTE transaction which updates X (at which point the lock on X is released), but before the transaction envelope of *task_B* can lock X and Y, another external activity C can start. The activity of C is also shown in figure 4.6. C can finish quickly enough so that the transaction envelope (of *task_B*) can successfully lock X and Y, and then go on to do the update X. Since X has actually been changed by C already when *task_B* uses it, this results in a "incorrect summary" on X.

Figure 4.7 shows the inadequate protection against external activities when transaction envelopes are used in a nested Pern transaction (for a task segment). Here, a PCTE transaction (on behalf of an external activity) that starts and finishes between the time of two Oz activities could potentially cause the incorrect summary problem.

A solution to this problem is to have a **lock mediator** process to lock all the objects needed for a Pern (nested) transaction. As shown in figure 4.8 (the dashed lines represent correspondence between Pern transactions and lock mediators), one instance of such a mediator process is needed for each top level Pern transaction. Our implementation of the lock mediator is the following: when the Oz server is started, also start the master lock mediator process and set up a pipe (Unix FIFO) between it and the Oz server process. Whenever a top level Pern transaction is started, Oz (on behalf of Pern) notifies the master mediator process to fork and spawn a child mediator process for this new transaction. The master mediator also sets up a named pipe, identified by the top level transaction id, between it and the child mediator process. Whenever a Pern transaction needs to lock or unlock an object, it notifies the master mediator process of its top level transaction id, the object id and the action (lock or unlock), using the pipe between the Oz server and the master mediator process. The latter then sends the lock/unlock request to the corresponding child mediator process using the named pipe between these two processes. The child mediator process does the actual lock/unlock for objects in the PCTE objectbase. It also starts a PCTE transaction when

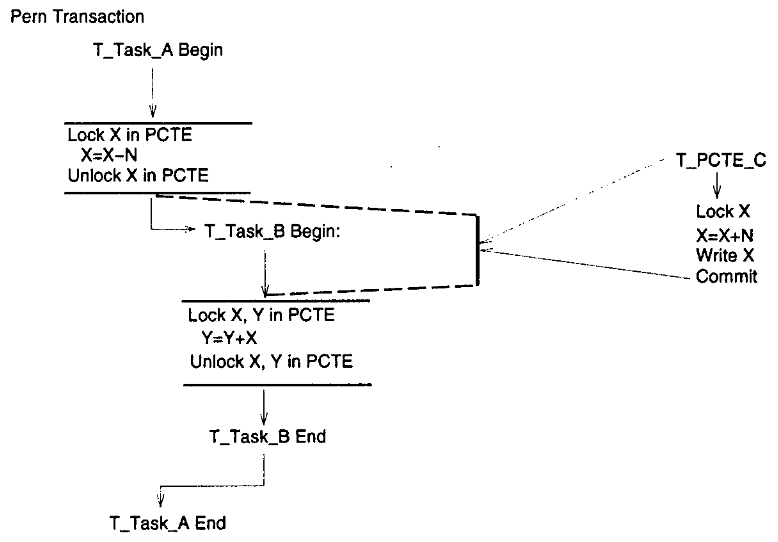


Figure 4.7: Incorrect Summary Problem

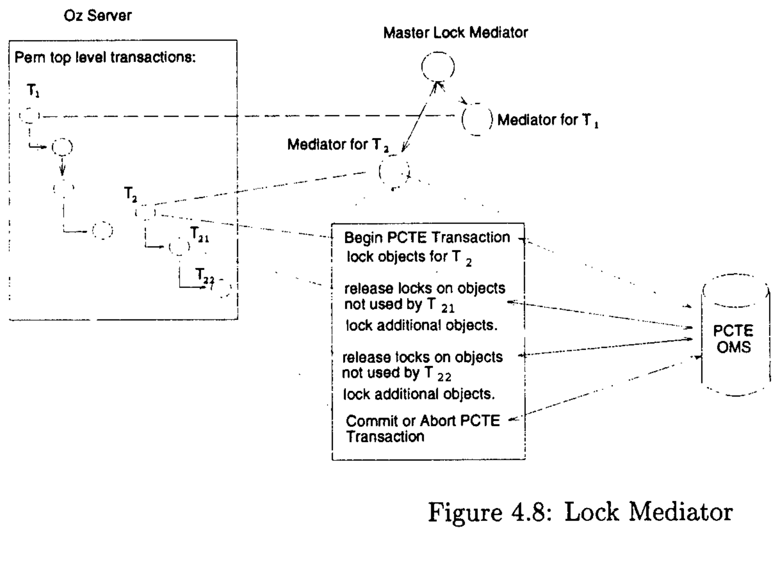

Oz Server
Pern top level transactions:
T1
T2
T21
T22
Master Lock Mediator
Mediator for T1
Mediator for T2
Begin PCTE Transaction lock objects for T2
release locks on objects not used by T21 lock additional objects.
release locks on objects not used by T22 lock additional objects.
Commit or Abort PCTE Transaction
PCTE OMS

Figure 4.8: Lock Mediator

the top level Pern transaction starts, and commits the PCTE transaction when the top level Pern transaction is to commit. Copying object contents from/to PCTE should also be done by the child mediator when it locks/unlocks an object. Any failed lock request to PCTE will trigger the child mediator to abort the PCTE transaction. It then notifies Pern, through the master mediator, to abort the top level Pern transaction. Using this scheme, each top level Pern transaction, along with its possible subtransactions, correspond to a single PCTE transaction. Locks can be transferred between Pern transactions that belong to the same top-level transaction because all the locks within a top-level transaction are held by a single process.

Comparing to the transaction envelope approach, the advantage of using a lock mediator is that it actually maps each top level Pern transaction to a PCTE transaction and can therefore guarantee the integrity of the Pern transactions. The disadvantage is that it may be relatively heavy weight since one child mediator process is created for each top level Pern transaction. The choice between the two solutions may depend on the actual application environment and the degree of protection that is required. If it can be assumed that most of the objects used by Oz would not be accessible to other PCTE activities, for example, by setting the proper group permissions on the objects (the Emeraude PCTE has commands for setting and changing user/group permissions), then using transaction envelopes is probably adequate. Otherwise, if Oz intends to use the objects that are shared by other PCTE activities, the lock mediator is in order.

We did not fully implement the lock mediator approach due to time and resources constraints.

4.7 An Example: Oz/Doc with PCTE OMS

We now demonstrate an example of using Oz and the PCTE OMS together. We ported an existing Oz environment, Oz/Doc, to test the modified Oz server (which interfaces with the PCTE OMS), and to conduct some performance analysis.

4.7.1 The Oz/Doc Environment

Oz/Doc is a document production environment implemented on top of Oz. The document composition structure (for example, a manual is composed of a set of submanuals, each of which consists of a header, a set of chapters and figures) is represented using an Oz object-oriented data model. A set of Oz task definitions was employed to implement the process of document production (for example, first *reserve*, then *edit*, *format*, *deposit* and *print*).

We chose this environment as a test case in our experiment because the data model of Oz/Doc is relatively simple and thus it can be mapped into the PCTE OMS easily. Another motivation is that there are readily available PCTE tools, for example, the "Edit Object" command in the OMS Browser, that can be used as concurrent PCTE tools (with regard to the Oz tasks in Oz/Doc, e.g., *edit*) to test our concurrency control mechanisms.

In our experiment, both the control data (Oz process data) and the product data are stored in the PCTE OMS. However, end-users need not be aware that the PCTE OMS is in use. They normally work in an Oz environment using the Oz Client GUI, which is responsible for sending object and task requests to the Oz server. The Oz client interface includes an objectbase display which shows the entire objectbase hierarchy. Users can "navigate" the objectbase (and thus "zoom in" to any part of the objectbase) by clicking the objects in the display. Tasks and Oz built-in commands can be selected from the menus. Clicking an object identifies it as a parameter for a selected task. External tools (for example, *emacs*) are encapsulated as the activities of Oz tasks. They are invoked by the Oz clients and thus normally run on the same workstation where the Oz client was started.

```
HISTORY  :: superclass ENTITY;
      history : text;
end

TIMESTAMPED :: superclass ENTITY;
 time_stamp : time;
end

VERSIONABLE :: superclass ENTITY;
         version_num : integer = 0;
         state       : integer = 0;
         locker      : user;
         reservation_status :
          (CheckedOut, Available, None) = None;
         version     : text    = ",v";
end

FILE :: superclass TIMESTAMPED, VERSIONABLE,
                  HISTORY, ENTITY;
      owner    : user;
      contents : text;
end
```

Figure 4.9: Definition of the FILE class (with multiple inheritance) in Oz/Doc

4.7.2 Implementing the Oz/Doc Environment with Oz and the PCTE OMS

To implement the Oz/Doc process, we first map a Oz/Doc data model onto the PCTE OMS model using the approaches discussed in section 4. Most of the data definition mappings are straightforward. Figure 4.1 and figure 4.2 give an example on how to define an Oz object class in the PCTE OMS. Oz/Doc has only one class definition, *FILE*, that is derived from multiple super classes, namely, *TIMESTAMPED*, *VERSIONABLE*, *HISTORY*, *ENTITY*, as shown in figure 4.9. The multiple inheritance is only one level. Therefore, as shown in figure 4.10, we simply duplicated the attribute definitions of the first three super classes in the class definition for *FILE* and make it a subclass of *ENTITY*.

In the Oz/Doc environment directory, there are two files that need to be set up specifically when the PCTE objectbase is in use. Figure 4.11 shows the contents of these two files. For *objectbase*, it specifies the root (top level) object for all Oz objects (of Oz/Doc) stored in the PCTE objectbase. It is from this root Oz object that the *read_objectbase* function in Oz server starts the objectbase traversal. The *strategy* file usually contains the compiled Oz data model and task definitions. In our Oz/Doc environment, it first specifies the names of the schemas that make up the working schema set. Then it lists the name of the schema, oz_doc, that defines the Oz/Doc data model (figure 4.10 is actually part of the output of the PCTE command "sds_decompile oz_doc", which prints out the definitions of a PCTE schema). *sys* and *env* need to be in the working schema set because *oz_doc* references object types defined in these two schemas. The *read_strategy* function in the Oz server uses these schema names to set the working schema set, and then reads the class hierarchy from the PCTE OMS according to *oz_doc*.

Once the data model for Oz/Doc and the object instances are loaded into the Oz server, users can start working in this environment via the Oz clients. For example, say a user can start the *edit* task on a *FILE* object named *introduction* (for example, by selecting the *edit* command off the

```
FILE__time_stamp : integer := 0 ;
FILE__version_num : integer := 0 ;
FILE__state : integer := 0 ;
FILE__user : integer := 0 ;
FILE__locker : integer := 0 ;
FILE__reservation_status : string := "None" ;
FILE__owner : integer := 0 ;

ENTITY : subtype of object ;
FILE : subtype of ENTITY ;

FILE__history : composition link to Oz_FILE;
FILE__version : composition link to Oz_FILE;
FILE__contents : composition link to Oz_FILE;

extend FILE
  with
  attribute FILE__version_num ;
    FILE__time_stamp ;
    FILE__state ;
    FILE__locker ;
    FILE__reservation_status ;
    FILE__owner ;
  link  FILE__history ;
        FILE__version ;
        FILE__contents ;
end FILE ;
```

Figure 4.10: Corresponding Definition of FILE in PCTE (using single inheritance)

```
objectbase:          strategy:

_/OzDoc.sys          PCTE_BEGIN
                     sys|oz_doc|env
                     oz_doc
                     PCTE_END
```

Figure 4.11: Objectbase and Strategy Files

menu and identifying *introduction* by clicking it on the objectbase display). Oz copies the contents of *introduction* from PCTE to a Unix copy and creates a backup copy for it, before invoking the Unix edit program on the Unix copy. It also copies the Unix copy back to PCTE if the task succeeds (the governing Pern transaction commits), or the backup copy back to PCTE if the Pern transaction rollbacks. Both the Unix copy and the backup copy of *introduction* are removed after the *edit* task (and the appropriate commit or rollback actions) completes.

To the users, all these file I/O operations happen "behind the scene" in the sense that the users are not aware at all that the PCTE OMS (instead of the native Oz OMS) is in use.

4.7.3 Performance Analysis

To get some idea of how file I/O operations has affected the performance of task activities, we defined a *file_io* task with an activity that concatenates "hello world" to the end of a file. We then used an Oz tty client, which can execute tasks in batch mode, to fire this task on the same file (a PCTE object) 100 times. The average total time it took to finish all 100 task invocations was 316 seconds. Using the same task in an identical Oz/Doc environment where the Oz OMS is in use, it took an average 115 seconds to finish all 100 task invocations. From the discussion in the previous section, we see that the following additional file I/O operations were performed when the PCTE OMS was in use: file creation in Unix, file copied from PCTE to Unix, file copied from Unix to PCTE, and file deletion in Unix. The two copy operations were responsible for the bulk of the difference in total elapse time.

It is evident from this result that when the PCTE OMS is in use, file-related Oz tasks will take longer to finish (from a user's perspective) because of the additional file I/O operations. This performance hit is inevitable.

4.8 Related Work

HyperWeb [69] is a framework that supports the construction of hypermedia-based software development environments. In Hyperweb, software artifacts is stored in the PCTE OMS. Moreover, the hypermedia linking of these software artifacts are also modeled and stored in the PCTE OMS – each hypermedia node is an object and each hyperlink is a relationship between two objects. The (PCTE) working schema set is used to constrain the types of links that can leave or enter a node. The architecture of HyperWeb is client/server. The HyperWeb server communicates with the PCTE OMS, which acts as a software object base, providing data integration. The server coordinates tool activities among clients through message passing. HyperWeb is very similar to Oz in using the PCTE OMS for data integration. Both have the client/server architecture where the server is responsible for communicating with the PCTE OMS. Also inferred from the HyperWeb paper is that the server and clients run outside of the PCTE environment. There was no discussion of concurrency control or process integration.

The CORBA, ATIS and PCTE integration [6] was an interesting experiment to test the possibility of coupling CORBA, PCTE and CASE tools. It layered CORBA on top of PCTE so that CORBA provides the interfaces for external access to the underlying object model implemented in the PCTE OMS. To guarantee the interoperability between CORBA and PCTE, the CORBA distributed and secure execution services were implemented to conform with the PCTE Execution and IPC specification. Some CASE tools, i.e., the ATIS (A Tool Integration Standard) Version and Configuration Services, were added to this hybrid environment to demonstrate the advantages of the distributed object (CORBA) interface on top of PCTE in terms of data and control integration.

4.9 Conclusion

Our approach might be considered as a light integration with PCTE because the Oz server, Oz client and external tools remain as alien processes to PCTE. Theoretically we could have put all the Oz code into the PCTE objectbase so that Oz runs within the PCTE environment, but given the sheer amount of source code and the heavy Unix dependencies, we did not believe a complete porting was practical. We believe that our situation is typical for a legacy SDE that is adapted to a new framework. Because the Oz components were loosely coupled and the PCTE OMS provides a standard API, we were able to interface Oz with the PCTE OMS, thus accomplished the integration between the two.

Our experiment has successfully enabled Oz to interface with the PCTE OMS. The resulting system is an environment where software tools can be integrated by sharing the same Oz process and the same PCTE objectbase. From the point of view of integration technology, we showed that Oz and PCTE are complementary and add value to one another. For tools running in an Oz environment, using the PCTE OMS enables them to also share data with other PCTE tools (that are not in the Oz environment). For the PCTE environment, using Oz allows process integration among Unix tools that access the PCTE OMS. This can be very useful since not all software tools will be "tightly" integrated into PCTE (moved into PCTE); rather, they will still run on Unix while using the PCTE OMS and other services.

This experiment shows that the Oz server can be modified to work with an external OMS with a somewhat different data model. A better part of this experiment was devoted to the concurrency control problem, where we wanted to devise a mechanism to incorporate PCTE transactions into the more flexible Pern transactions. This is necessary to prevent the tools written to use the PCTE interface directly from avoiding our policy enforcement mechanism. The transaction envelope is a light-weight, clean but a bit simple-minded solution. The lock mediator solution is more elaborate and sophisticated but heavy-weight.

4.9.1 Lessons Learned

Since Oz already had a fully functional process engine and a certain degree of abstraction in terms of object access, we were able to focus on the making an interface to the PCTE OMS in order to build an environment that has both process integration and (open and standard) data integration services.

The limitation that one Unix process can have only one PCTE transaction, coupled with the current Oz architecture (one Pern process per Oz server), certainly made it difficult to elect a simple and yet complete solution to implement Pern nested transactions. Obviously if PCTE allows nested transactions within a single Unix process, mapping (starting) a PCTE transaction for each Pern transaction becomes an easy solution to this problem. Alternatively, Pern can be re-engineered into a separate program; and in run-time, a Pern process is created (spawned) for each top-level Pern transaction and locks (PCTE) objects in a scheme similar to the lock mediator approach. This later solution is much more complicated.

Finally, as mentioned throughout this chapter, there were fixed assumptions about the OMS in the earlier versions of the Oz code. These fixed assumptions were eliminated during later phases of the work reported here. All access to class, object and attribute are now done through function calls to an OMS API. Had this interface been in place when we started our experiment, we could have just implemented a set of functions that can be used by the API to access the PCTE objectbase. However the major data model mismatch (for example, files are component objects instead of object attributes) still requires changes outside of Oz's OMS interface (for example, in the process engine if the code assumes only *file* attributes, it needs to be changed to also consider *composite* attributes of *Oz_FILE* if the PCTE OMS is in use).

4.9.2 Future Work

Pern and and more recently, a new Process Engine called Amber, have been developed as independent components. Although re-implementing the entire Oz system in PCTE is not practical, it is now feasible and would be interesting to experiment putting (re-implementing) these components into PCTE. Putting Pern into PCTE would add support for flexible transaction models to PCTE.

Further, its mediator interface would enable PCTE environment builders to tailor Pern to suit their environment-specific concurrency control policies [107]. Similarly, putting Amber into PCTE would add process integration, not addressed in this chapter. Amber has a mediator interface that enables environment developers to tailor it to support environment-specific process enforcement and assistance models. Since our experiment is a light integration, it is difficult for Oz to enforce controls on tools running within PCTE other than using data locking, which provides only passive control. However if the Oz components are put into PCTE, they can take advantage of the control integration services provided by PCTE, e.g., the message queue, support for user roles, and security control, to provide active control and integration for the PCTE tools.

Chapter 5

Integrating Synchronous Groupware

Abstract

Computer supported cooperative work (CSCW) has been recognized as a crucial enabling technology for multi-user computer-based systems, particularly in cases where synchronous human-human interaction is required between geographically dispersed users. Workflow is an emerging technology that supports complex business processes in modern corporations by allowing to explicitly define the process, and by supporting its execution in a workflow management system (WFMS). Since workflow inherently involves humans carrying out parts of the process, it is only natural to explore how to synergize these two technologies. We analyze the relationships between groupware and workflow management, present our general approach to integrating synchronous groupware tools into a WFMS, and conclude with an example process that was implemented in the Oz WFMS and integrated such tools. Our main contribution lies in the integration and synchronization of individual groupware activities into modeled workflow processes, as opposed to being a built-in part of the workflow WFMS.

5.1 Introduction

Human-to-human collaboration and coordination is critical in any multi-person product development effort. In cases where the work requires intensive use of computers, computerized support for collaboration is essential, particularly when the collaborating users are physically dispersed, a scenario that is becoming more common with the recent advances in networking technologies and the growing popularity of the Internet and the World Wide Web.

Workflow management is an emerging technology that is concerned with modeling and executing *business processes.* As defined in the workflow coalition model [110], a business process is "a procedure where documents, information or tasks are passed between participants according to defined sets of rules to achieve, or contribute to, an overall business goal". A WorkFlow Management System (WFMS) thus provides a formalism (e.g., Petri nets, task-graphs) in which processes are defined, and a corresponding workflow engine in which processes are "executed", where forms of execution include automation in scheduling and activating activities according to the defined process; reactively triggering activities based on state changes; monitoring the process; and enforcing policies and consistency constraints (e.g., on the data being accessed during the process).

Since workflow inherently involves multiple humans carrying out parts of the process, it is only natural to explore how to synergize groupware and WFMS. This chapter investigates support for defining groupware activities in a process, and executing them as part of an on-going workflow. Our focus is on integrating individual (external) synchronous tools, such as multi-user editors (e.g., Flecse [53]) and virtual whiteboards [115], into a process executed in a WFMS. This is in contrast to interfacing the WFMS framework with entire CSCW development toolkits or environment frameworks (e.g., ConversationBuilder [130]).

The possible degree of integration of groupware tools into the WFMS lies in a wide spectrum. The simplest method involves "inserting" a single tool as an isolated entity in the process and invoking it using the same notations and mechanisms as for regular (single-user) tools. This is obviously a very limited form of integration; for example, it does not supply mechanisms to identify and bind the participants to the execution of the activity from within the workflow framework, and it does not allow to associate the activity with other related activities. At the other end of the spectrum, a groupware activity may be fully integrated in the WFMS by becoming an undistinguished part of the WFMS framework *itself,* as opposed to being part of a particular process that is enacted on the WFMS. Such an approach is taken by various existing WFMSs (e.g., Lotus Notes [134]), in which the WFMS is essentially treated as a CSCW system. While useful in its own right, it does not address the need to integrate external tools which are defined as part of a specific process description, and for which the WFMS does not have its own "native support". This leads to our approach, which is process-specific tailoring and integration of groupware activities. This approach requires means to embed groupware tools such that it is possible to define control-flow, constraints and other rules of invocation for the activity, and to supply additional notations and mechanisms for handling multi-user synchronous interactions. Here again, there are several levels of integration of the tool with the underlying framework, ranging from being black-box where the internals of the tool (e.g., source code) are not accessible to the WFMS, to "gray-box" integration if the tool provides some application programming interface, to "white-box" integration. While white-box integration may have a potential for higher-degree of integration, it cannot be employed when there is no access to the tool's source code, or when modification of the (external) tool might be too difficult. Hence, our focus in this chapter is on models and mechanisms that provide for full integration of individual groupware activities as units of a workflow process, but treating the tools themselves as encapsulated entities.

5.1.1 A Motivating Example

Consider a workflow for reviewing documents (e.g., research paper, or a business plan) by a group of independent and physically-dispersed experts. A `Review` task, illustrated in Figure 5.1, may be defined as follows. A coordinator sets-up the team of reviewers, by possibly running a `setup-review`

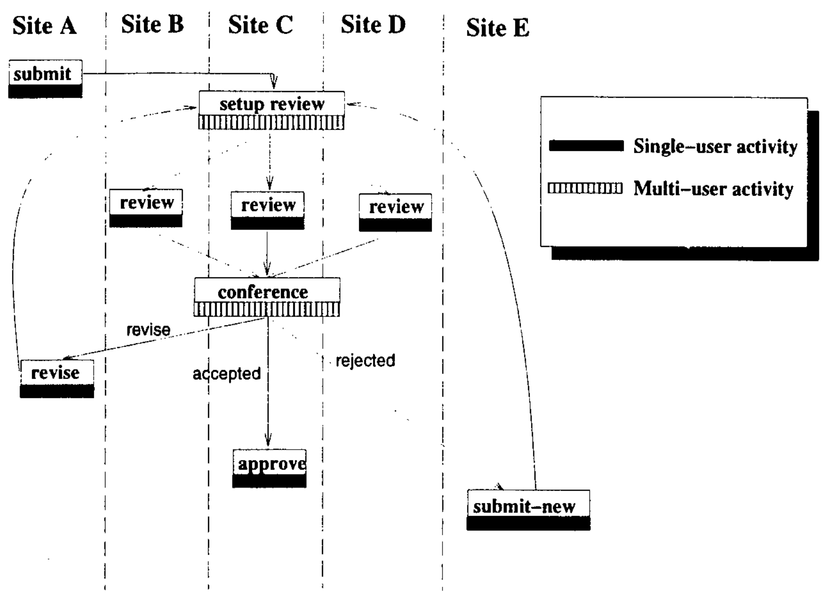


Figure 5.1: The Review Task

tool that identifies and selects a qualified set of experts who agree to review the document. Next, they review the document individually. Once they all complete, a multi-user virtual conference meeting takes place, where they discuss together the document and possibly reconcile their conflicts. After the meeting, if the document is accepted, the coordinator completes the review task and proceeds with the approval task. Otherwise, if the document needs revisions, a revision request is sent to the author(s) of the document, after which the task reiterates to the personal review phase. If the review concludes that the document is totally unacceptable in its present form, it is rejected and a new submission (perhaps from another person) is requested. Each of these activities may be associated with an automated set of actions, or enforcement rules. For example, the **approve** action for paper review may entail notifying the author and the publisher and sending the proper author kit (or proceeding with actions to deliver the requested venture capital in case of a buisness plan review).

In order to support such task, the WFMS should have the ability to: (1) *define* such task using the WFMS process modeling formalism, including definition of the groupware activities, local and global constraints on their activation, and local and global inter-activity control-flow and "user"-flow; and (2) *execute* such task, including mechanisms that automate the control- and user-flow, enforce the constraints, and in general enabling groupware activities to affect, and be affected by, other local or groupware activities from same or from different users.

The rest of this chapter is organized as follows: Section 5.2 discusses the relationships between groupware and workflow, and overviews related work; Section 5.3 presents our approach to enabling integration of groupware activities in a process; Section 5.4 outlines the realization of the approach in the Oz WFMS and presents an example Oz (sub) process that integrates groupware, and Section 5.5 summarizes the chapter.

5.2 Relationships between Workflow and Groupware

Although both fields deal with certain common issues and overlap with each other, their orientation is quite different, and it is important to realize these differences as a basis for understanding both the need for, and the general approach to, our synergy. Workflow management in general focuses on support for *process*, including its explicit representation and executability, involving users, tools and artifacts. Further, it is concerned with allowing to specify and preserve the consistency and integrity of the process and its related artifacts (e.g., documents, products, etc.), particularly for tasks that require such support. Finally, it typically involves integration of single-user, or asynchronous multi-user tools (in fact, the WFMS may itself be viewed as such "tool").

CSCW, to the contrary, is less concerned with (formal) processes, and is mainly concerned with human-human interface and inter-personal cooperation and collaboration paradigms. Process is incidental and implicit, and therefore unsupported. Thus, while both technologies are geared towards supporting collaboration among people in organizations, they are mostly complementary. Workflow technology can benefit from integrating groupware by embracing the human-human interaction paradigms and tools, and groupware could benefit from workflow by adding explicit and consistent process definition and enactment.

Although groupware in general refers also to asynchronous tools such as electronic mail and electronic bulletin-boards, in this chapter we focus on the classical *synchronous* groupware tools, in which multiple users collaborate in a virtual shared space, also known as "same time, same place" technology [89]. Each such tool is invoked for or otherwise affiliated with a designated set of users all at the same time and they all terminate their connection at the same time (modulo network delays and other implementation-specific glitches). We have in mind tools ranging from multi-user editors and debuggers (e.g., the Flecse toolkit [53]) to document inspection systems (e.g., Scrutiny [85]). Most significantly from the viewpoint of workflow, synchronous groupware requires special integration facilities that do not exist in conventional WFMSs, and at the same time, once they are in place, there are many automation opportunities and control dependencies that can be associated with their activation, which is the main reason for focusing on them. (An asynchronous multi-user tool enveloping approach that allows the WFMS to submit multiple activities to the same "persistent" tool invocation is described elsewhere [227].)

There have been some systems that attempted to combine workflow and synchronous groupware. One such system that originated from CSCW is Conversation Builder (CB) [130]. The main concept in CB is that of a *conversation*, that serves as a context in which a user performs its actions ("utterances"), and can potentially affect other users participating in the same conversation through a shared conversation space yet still protect their private conversation space. In addition, CB enables to specify activities and their interrelations using *protocols*, which are state-machine descriptions of the flow of the conversations, or in other words, a limited form of process modeling. However, it has no process enactment engine.

Scrutiny [85] is a code-inspection system built on top of CB that supports a specific methodology — Fagan-style code inspection [65] — translated into process. This includes support for different roles (moderator, author, etc.) and ensuring that the inspection follows the defined process, in addition to the underlying groupware framework for supporting synchronous inspection among multiple remote users. Scrutiny is not a generic WFMS, however; the workflow process is built-in.

On the WFMS side, most systems provide some degree of "groupware", or multi-user support because they are inherently multi-user. However for the most part the support is either built into the WFMS (as opposed to integrating external tools), or it supports asynchronous external groupware. Examples include InConcert [44], ActionWorkflow [70] and ProcessWEAVER [67].

Finally, there is yet another approach to supporting collaboration that is orthogonal to both workflow and groupware, namely (collaborative) concurrency control. Users going about their own business happen to try to access the same data in conflicting ways (e.g., two writers). The concurrency control mechanism *reacts* to such attempts, and typically disallows all except one of the accesses. We have conducted extensive research on semantics-based concurrency control, as have others, and there are many proposed approaches that provide collaboration by permitting "conflicting" accesses when they happen to come up, perhaps with an attached "resolution" procedure (see [16] for a survey). We take here a complementary *proactive* approach to supporting collaboration.

5.3 Integration Concepts and Mechanisms

5.3.1 User Modeling

In order to identify and specify explicitly which users should be assigned to the execution of an activity, users must be somehow represented in the system. The simplest way to represent users is by their operating-system (or any other system-supplied) id. This allows the process to associate users with activities, either by directly specifying the id, or indirectly via a "user" attribute that is dynamically bound to such id at runtime. We defer the discussion of static vs. dynamic binding of users to activities to Section 5.3.2; regarding the representation, this approach is clearly limited. For example, it would be impossible to associate users to activities based on general characteristics and specific "state" of the users, such as their roles, whether they are active in the system, their physical and virtual location, etc. Moreover, the lack of such attributes may not allow the workflow engine to assist in the activation of groupware tools by, for example, locating available and qualified users.

Thus, a more suitable representation of users should be provided. Our approach is to model users as a distinct entity, much in the same way that typical WFMSs model data, process, and tools. In other words, in addition to data-, tool- and process- (or control-) integration, we employ user-integration. This leads to abstracting users as objects in a user-repository. More specifically, each user is represented as an object that stores pertinent information that is needed by the operating process, and is instantiated from a primitive *user* "base" class, or one of its derived sub-classes. Individual user objects can be aggregated in a *user-group*, which can be used for three different purposes: (1) represent an organizationally determined sets of users, such as members of a project team (an "is-part-of" composition hierarchy); (2) classify users based on common characteristics, such as their "role" ("is-a" class hierarchy); and (3) represent a set of users that is grouped specifically for the purposes of activating specific groupware tool(s). Note that we deliberately use the more general "group" term and avoid typical built-in mechanisms to define roles (which, as recognized by [99], can sometimes be as much limiting as assisting), although such a notion can be imposed on top of the generic user and group classes.

Once users and groups are modeled as objects, they can be pointed-to by other kinds of objects. For example, a multi-author document object can point to its "owner" object(s), and one user object can point to another "manager" user-object. This enables to query the user repository in conjunction with the artifact repository in order to, for example, automatically assign the review of a given document to its author(s) and notify their manager(s). But in order to provide such functionality, the repository must supply a mechanism that enables the process to select users based on the values of their attributes. The WFMS should strike the right balance between providing built-in support for few mandatory attributes of these classes, and allowing to extend these class definitions with optional, process-specific attributes that are defined and manipulated on a per-process basis.

To illustrate the concept of user modeling, consider the following sample *user* and *group* classes defined in the Oz data modeling language, shown in Figure 5.2. In this example, users are represented by the `USER` class. Each object that gets instantiated from that class has several "state" attributes (e.g., host from which user is connected, which is in general different from the host in which the `USER` object resides due to the client-server architecture, see Section 5.4), and a set of links to `USER_GROUP` instances. A WFMS supporting such class may elect to provide built-in support for none, some, or all of the attributes in that class. For example, when a user logs-in to the WFMS, the WFMS may automatically attach the user to his proper `USER` object by looking up the `userid` attribute, and subsequently fill-in some values for the built-in attributes based on the login information. The current version of Oz supports only the attachment to `USER` object as a built-in facility (actually, even this is not strictly enforced, since an Oz environment may have no user modeling at all in cases that it is unnecessary), although any other attribute can be defined and manipulated by a specific process.

```
USER_GROUP:: superclass PROTECTED_ENTITY;
        name : string;
        users: set_of link USER;
end
USER:: superclass PROTECTED_ENTITY;
        id :         userid;
        name:        string;
        host_name: string;
        host_ip:     string;
        active:      boolean;
groups:     set_of  link USER_GROUP;
end
```

Figure 5.2: **Sample User and Group Classes**

5.3.2 User Binding

Once users and groups are properly modeled in the system, the process definition language must enable to associate user objects with the activities. This binding method depends on the underlying user-interaction and user-control models that WFMSs employ. In "active" WFMSs the process executes on behalf of the "system", in which case any (at least any interactive) activity must be assigned to some user, or to a set of users in case of a multi-user activity. In contrast, in "reactive", user-driven WFMSs, each activity is by default executed on behalf of the user who invoked it. In this case, single-user activities may still need to be *delegated* to other users. For example, ProcessWEAVER [67] supports *agendas*, which are personal "to do" list that can be updated by the WFMS or by other users for delegation purposes. Although delegation can be considered as a (somewhat restricted) form of groupware, we do not discuss it here; see [21].

The simplest method to bind users to activities is to specify them (via their object representation) directly in the process model as the "recipients" of the activity. The main problem with this static approach is that in order to change the binding set, the process model has to be modified and consequently the instantiated process must be evolved, a non-trivial task [73]. This is particularly evident in cases where the same activity could be bound to different users depending on the context in which the activity is invoked and other time- and location-dependent circumstances.

Our approach is to provide *dynamic* (late) user binding. When the process is initially defined, groupware activities are associated with *classes* of qualified users or groups, as opposed to instances. At run-time, they are attached to an activity based on both their class membership as well as the particular values of their state attributes. A group of participants can then be formed by either binding directly all (active) members of a given class, or by binding a set of individual users (which are not necessarily all part of the same group) which satisfy a certain condition.

By modeling users as objects and employing dynamic binding, a process can utilize the WFMS's regular data-query facilities in order to *select* the proper users based on the knowledge stored in these objects. For example, the process may be able to select only users that are known to be active, denoted by having a `true` value in the `active` attribute of their user object. Furthermore, if the process modeling language supports the notion of pre-condition or "guard" predicates (as many do), it can be applied to the user binding-set to check or verify that the selected user-set and its cardinality are valid for the activity. Finally, if the WFMS supports automatic invocation of activities to satisfy a failed condition (backward-activation), or following state-changes (forward-invocation), it can further assist in the process of locating qualified users. For example, a failed user-binding predicate could trigger an activity that can lookup, periodically poll, or otherwise employ a "wakeup" procedure to locate users.

Figure 5.3 shows an example of how the document-owner association mentioned earlier can be modeled in the Oz process modeling language (we ignore details of the language that are not relevant to the issue of user binding). The `multi-edit` activity takes a single document object as input (line

```
1) multi-edit [?d:DOCUMENT]:
2)  # OBJECT BINDING
3)  (and
4)     # bind users to activity
5)     (forall USER ?u suchthat (and
6)                          (linkto [?doc.owner ?u])
7)                          (?u.active = true)))
8)  # bind  relevant documents
9)  (forall DOCUMENT ?related suchthat
10)                  (linkto [?doc.ref ?related]))):
11)  # CONDITION
12)  # check that documents are allowed to be read by all
13 ) # users in the binding set
14)  (?doc.allowed_edit = ?u):
15)  # USER BINDING
16)  participants[?u]:
17)  # EXECUTE
18)   MULTI_EDIT multi_editor ?doc ?related-docs
19)  # ASSERT EFFECTS
20)  #0. document changed.
21)  #   this assertion may trigger other activities.
22)  (?doc.status  = Changed);
23)  #1. document unchanged.
24)  (?doc.status  = NotChanged);
```

Figure 5.3: **multi-edit activity**

1). It then binds to the activity all users that satisfy two conditions (lines 4-7): they must be owners of the document (modeled as a link from the document to the owner object), and they must be active (as denoted by the `active` attribute). After binding other related documents (lines 9-10), the activity checks whether the users in the binding set are allowed to edit the document (line 14); if this is not the case, the activity is aborted. Otherwise, the user-binding phase actually takes place (denoted by the `participant` directive in line 16), followed by invocation of the multi-user tool (line 18) followed by assertions that reflect the result of applying the tool (lines 20-24). The actual interface from the modeling language to the tool is done via enveloping mechanism, which is beyond the scope of this report, see [86].

Notice that depending on the selected document, each time the activity is invoked it would be assigned to the appropriate owners. Moreover, if the owners of the document change over time (this would be reflected by changing the `owner` links of the document), subsequent invocations of the same activity on the same document will automatically bind the new (active) owners.

Two additional closely-related issues to discuss regarding user-binding are: (1) "user-overflow", i.e., when the user binding-set contains more users than required by the activity; and (2) "user-underflow", when the user binding-set is smaller than required, either because there are no available users, or some users do not want to participate in such activity. In either case, some additional semantics must be associated, either by default and/or specified by additional syntax. Regarding overflow, the process modeling language needs to provide directives for subset-selection. Three plausible methods may be (1) interactive, i.e., the system prompts the coordinator who initiated the activity to make the selection; (2) random; and (3) following some priority scheme. The problem with user-underflow is more severe, since it effectively disables the execution of the activity, and therefore some failure semantics must be attached and further actions may be taken. We have identified the need for a `notify` action that directs the WFMS to notify (e.g., by e-mail) potential participants about future invocations, perhaps with a later `retry` action that actually re-invokes the activity. Alternatively, if the activity must be executed instantly, the WFMS may seek means to automatically activate users by invoking a triggering activity that would search for, and locate, qualified but currently inactive users.

5.3.3 Process Automation of Groupware Activities

Process automation lies in the heart of workflow management. Regarding automation groupware activities, we have already mentioned one form of automation, namely the ability to automatically satisfy a user-binding condition for setting-up a groupware activity. Another important capability that is required especially for groupware tools, is to be able to fan-in to and fan-out from the synchronous groupware activity and perform in parallel and asynchronously local and personal activities. This gives rise to distinguishing between globally defined, shared, multi-user activities, and locally defined, personalized activities, which each participant could define in his own private (sub-) process. In other words, if the process allows to define private workspaces with their own rules and state, the workflow engine could spawn multiple single-user activities as a result of, or as a preparation for, a groupware activity, without actually being required to know *how* these local activities are performed, and therefore delegating non-global definitions to the local users/sites. For example, in the Review task mentioned earlier, the `review` activity could be defined autonomously (and differently) by each user according to his own method of reviewing documents. This is the underlying motivation for the Summit model elaborated in [21] (although it actually deals with the general case of interoperating full-fledged processes). We will illustrate this control mechanism in Section 5.4.2.

5.3.4 Infrastructure Support

The above discussion made some implicit assumptions about the infrastructure support for groupware. This is in general open-ended and depends on the level of integration with the WFMS, but there are several basic requirements. First, the WFMS should be able to "locate" selected participants. This means that the system can locate and communicate with the client on whose behalf the participant executes (assuming a client-server WFMS as in the reference architecture [158]). Second, the WFMS should be able to (re)direct activities or parts of them across and among clients. Third, there must be a notification mechanism. Groupware activities, even if originating from one user, require to notify other remote users and ask them to perform the synchronous interaction. This setup procedure requires asynchronous mechanisms in which a client is notified and acts in reaction to a server request, which is the opposite of conventional client-server interaction. A related aspect is concerned with the user-interface for such notification. Such mechanism has to prompt the (perhaps unexpecting) user and attract its attention, and must allow the user the option to refuse to perform the action (or request to delay it).

5.4 Groupware Integration in Oz

We outline the implementation of the above ideas in the Oz framework and illustrate their use in an effective process for the Review task discussed in the introduction.

5.4.1 Oz Overview

Oz [21] is a multi-site collaborative WFMS that supports interoperability among heterogeneous and autonomous processes. Initially, it was designed to support software engineering projects (also known as process-centered software engineering environments, see [76] for a book surveying such systems), but later has been generalized to support workflow in various application domains (e.g., healthcare workflow [148]).

Oz introduces a flexible and dynamic mechanism to specify and integrate the desired interoperability between multiple process models (called the *Treaty* protocol), and corresponding execution support for multi-process activities that enables execution of activities with data/tools/users from multiple sites and fulfillment of their local prerequisites and consequences (the *Summit* protocol).

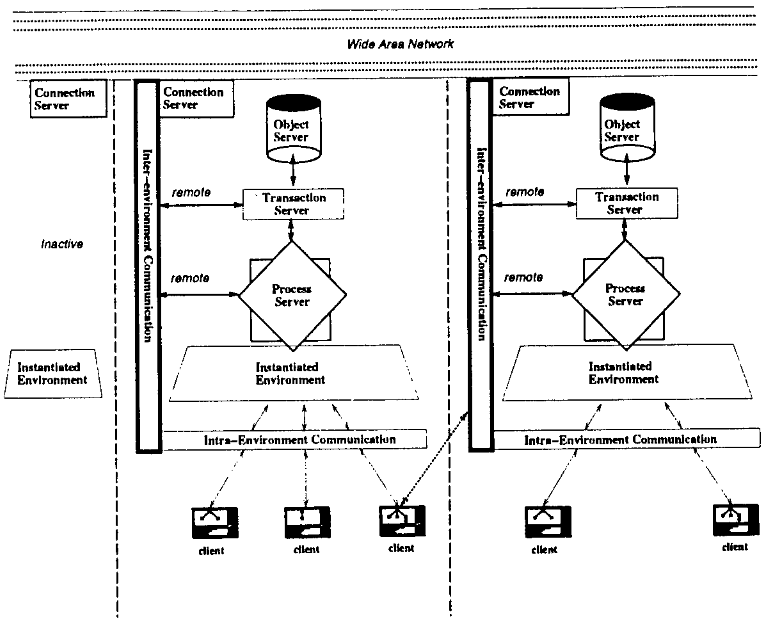


Figure 5.4: **Oz External Architecture**

The architecture of Oz is illustrated in figure 5.4; it is "shared nothing", i.e., the system is physically as well as logically decentralized, with each site maintaining autonomously its own project database, schema, process, and tool base, thus not limiting a priori the scale, scope or physical location(s) of the project being developed. Interactions among sites utilize local client-remote server and server-to-server connectivity facilities provided by the system (local client-local server connections assume a shared network file system, but need not reside on the same host).

Treaties, Summits and interconnectivity support are discussed elsewhere [21, 27, 26]; here we investigate collaboration among multiple users regardless of whether they work within the same or in different processes, or within the same local area network vs. across a wide area network. Nevertheless, decentralization and geographical dispersion were prime motivations for this work: we discovered quickly the limitations of ad hoc approaches when we had to deal with interactions among logically and physically distributed users.

Human interaction with the environment is through a client that provides the user interface as well as the workspace in which activities are executed. When a user issues a command (often indicating a rule, see below) he/she wants to perform, the request is sent to the server to check whether the activity can be executed (e.g., all prerequisites are satisfied and no parameters are locked exclusively by another user) and explores opportunities to automate and/or guide the execution of this or related activities (e.g., invoking other activities to attempt to satisfy the prerequisites or negotiating according to relaxed concurrency control policies [105]), and eventually returns to the client — either with the necessary information to execute the activity or to inform the client that the activity cannot be performed at this time.

A local process in Oz is defined using a rule-based language. Each activity is enclosed in a rule with formal typed parameters, and optional condition and effects that serve two purposes: to enforce and assert conditions that pertain to the activity itself; and to connect (through predicate/assertion matching) to other related activities and specify automation and/or atomicity requirements across activities. Related activities can be invoked automatically as part of either backward chaining to satisfy the predicates in a rule's condition, or forward chaining as a result of the assertions in a rule's selected effect (a rule may define more than one effect, but exactly one is asserted depending on the results of the encapsulated activity). A rule thus defines a process step, and the set of all chains emanating from that rule (implicitly) define a task.

Oz allows for dynamic (or late) binding of data to activities. The actual parameters to the rule are designated either explicitly by the user, or automatically by the process engine in case of a chained-to rule (see [108]). In addition, the language binds derived parameters in a *binding clause* that queries the project database, usually resulting in a set of objects structurally and/or logically related to the actual parameters.

Oz integrates most of the groupware facilities that were mentioned in the previous section, including user modeling, binding, automation, and infrastructure support for locating, setting-up, and notifying users through their clients. We now turn to the realization of the motivating example and show how the WFMS features can be exploited to support such process.

5.4.2 Realization of the Review Task

An Oz process that supports the Review task consists of: a schema (data model) with definitions for artifacts such as documents, reviews, revisions, etc.; a user model with proper `user` and `group` classes; envelopes encapsulating the tools, for example a conferencing tool [1]; and, most importantly from this chapter's viewpoint, a set of rules that specify how and by who the tools are enacted, their inter-dependencies with other rules, and potential for their automatic invocation.

Each of the boxes in Figure 5.1 could be realized as either a single rule, or as a set of interrelated rules collectively implementing the (sub)task. The edges between the boxes are represented by the matchings between effects of one rule and the condition of another rule. A rule is designated as either a single-user personal rule or as a multi-user, groupware rule (for pragmatic reasons the actual annotation is made in the tool definition as opposed to in the enclosing rule , but this is an irrelevant syntactic detail). If a multi-user rule is encountered, the rule-processor employs its infrastructure to select, locate, and connect to the proper participants; set up the activation and transfer control for the execution of the external tool; and regain control after the termination of the activity, setting the proper state values and possibly invoking other single- or multi-user derived rules.

Figure 5.5 shows two sample rules from the process (a full realization of a comprehensive multi-user "benchmark" process appears in [21]): the groupware `conference` rule and the personal `review` rule that precedes it. The `conference` activity binds all members of a group that is linked to the document (lines 3-6, 12); it is enabled only if a conference is requested on the document (line 13); it invokes the `conference` groupware tool (line 15); and it asserts one of the three possible outcomes of the conference that correspond to the three boxes in Figure 5.1: revise, reject, or accept (lines 16-21). Depending on the outcome of the conference, the proper rule will be triggered and assigned to the proper user. The `review` rule has a pre-condition that states that the document must be in a "reviewable" state (e.g. the author has completed it), and an assertion that either enables later invocation of `conference` (line 32), or disables it (line 34), depending on the outcome of the local `review` rule. (The `delegate` directive in line 29 is similar to the `participants` clause and binds the personal activity to a single user). Note that, as specified in the process definition, we want to enable `conference` only if and when `all` participants (i.e., reviewers) have completed their reviews. This is indicated by the `forall` universal quantifier (which is defined in line 7 but is actually used in line 13) that ensures the desired behavior (the latest version of Oz corrected this problem and the quantifiers are defined in the condition clause instead of the binding, as they should be).

5.5 Summary

The approach presented in this chapter shows how synchronous groupware activities can be synergized with workflow technology in a way that exploits the advantages of both worlds. By integrating such tools into a process framework, we enable to apply on them all the advantages that workflow provides: (1) Formal definition in the context of an enclosing process, and in specifying constraints and guidelines for user binding and invocation in general; (2) assistance in the execution of such

[1] In an actual implementation of a similar task we used the `white_board` public-domain tool, which enables multiple distributed users to share a virtual white-board on their screens [115].

```
1) conference[?d:DOCUMENT]:
2) (and
3) (forall USER_GROUP ?g      suchthat
4)                  (linkto [?d.reviewers ?g]))
5) (forall USER       ?users  suchthat
6)                  (member [?g.users ?users]))
7) (forall REVIEW     ?revs   suchthat
8)                  (linkto [?d.reviews ?revs]))
9) (forall DOCUMENT   ?rel    suchthat
10)                 (linkto [?d.related ?rel])))
11)
12) participants[?users]:
13) (?revs.status = ConferenceRequested)
14) # invoke the multi-user white_board tool
15) MU_TOOLS conference ?d ?u ?revs ?rel
16) # 0. enable revise
17) (?d.review_status = RevisionRequested);
18) # 1. no hope, go to reject. needs to start all over again.
19) (?d.review_status = Rejected));
20) # 2. review accepted.
21) (?d.review_status = Accepted);
###############
22) review[?d:DOCUMENT, ?r:USER]:
23) (exist REVIEW     ?review   suchthat
24)                      (and
25)                 (linkto [?d.reviews ?review])
26)                 (?review.owner = ?r.userid)))
27) :
28) (?review.status = ReviewRequested):
29) delegate[?r]:
30) REVIEW review ?d
31) # 0. enable review
32) (?review.status = ConferenceRequested);
33) # 1. indicate error
34) (?review.status = Error);
```

Figure 5.5: **Sample rules from the Review Task**

activities, by allowing the activities to modify the state of the process, thereby allowing to assist, automate, enforce, and monitor the activities as well as related (perhaps personal and asynchronous) activities.

The work described in this chapter mainly deals with essentially one process state. Expanding groupware to work across multiple processes with their own process state and "user space", running on true heterogeneous WFMSs, is a major future direction of this work.

Chapter 6

Integrating Asynchronous Groupware

Abstract

We present a tool integration strategy based on enveloping pre-existing tools without source code modifications or recompilation, and without assuming an extension language, application programming interface, or any other special capabilities on the part of the tool. This Black Box enveloping (or wrapping) idea has existed for a long time, but was previously restricted to relatively simple tools. We describe the design and implementation of, and experimentation with, a new Black Box enveloping facility intended for sophisticated tools — with particular concern for the emerging class of groupware applications.

6.1 Introduction

Process-centered environments (PCEs) and other task-oriented frameworks (see, e.g., the NIST/ECMA reference model [168]) usually support dialogues between external tools and the environment, which serves as a mechanism for integrating the tools according to their roles in the workflow. We identify three categories of integration methods, with respect to their approach to adapting the tools to the environment:

- *White Box*, where a custom tool is developed as part of a particular environment or a pre-existing tool's source code is modified to match a framework's interface. Custom tools may be prohibitively expensive to develop. Changes to pre-existing tools can often be implemented in a straightforward, repetitive manner, but nevertheless the source code must be available — perhaps an insurmountable difficulty when integrating off-the-shelf tools from independent vendors. The White Box approach is followed by several commercial message buses, most based on either the Field broadcast message server [193] or the Polylith software bus [188]. PCTE [221] and similar framework standards probably require more effort in tool adaptation, or a priori adherence to the standard by vendors, but enable a higher scale of integration. The CORBA interoperability standard [166] is not specifically directed to environment frameworks, and seems best suited to tools explicitly organized as servers — which relatively few are at present.

- *Grey Box*, where the source code is not modified but the tool provides its own extension language or application programming interface (API) in which functions can be written to interact with the environment. Relatively few tools, aside from database management systems, provide such convenience (although see [169]). Dynamic linking coupled with replacement of standard libraries (e.g., for I/O) works for some environments, e.g., Provence [140], concerned with monitoring simple events such as file system accesses, but it seems unlikely in the general case that arbitrary tools would happen to fit the protocols of a task-oriented framework. In particular, a PCE requires that task prerequisites be fulfilled prior to performing the task, so mechanisms to detect and/or notify that a task has already been completed are inadequate [181].

- *Black Box*, when only binary executables are available and there is no extension language or API. In this case, the environment must provide a protocol whereby *envelopes* extract objects and/or files from the environment's data repository, present this data to their "wrapped" tools in the appropriate format, and provide the reverse mapping for updated data and tool return values.[1] Envelopes may also be used in conjunction with Grey and White Box methods, but are mandatory for Black Box integration.

Our primary goal in this paper is to augment enveloping concepts and technology to apply to a much wider array of tools. We concentrate on the Black Box model, since it is often the only choice (particularly for legacy tools) as well as the most challenging.

Typical Black Box enveloping technology expects the tool integrator to write a script or program that handles the details of interfacing between the tool and the environment framework, often both to respect the environment's notion of task and to access its data repository, as well as the actual invocation of the tool with an appropriate command line and collection of any outputs and return values. In the case of a PCE, the process definition determines the workflow within which such a script or program may be executed. For example, the task's prerequisites may need to be satisfied in advance and its obligations fulfilled afterwards. The state of the on-going process execution usually sets the context for providing parameters to the tool and determines what should be done with its results.

This approach works well for tools, such as the standard UNIX toolset, that accept all their arguments from the command line at invocation, read and write some files (whose file system pathnames

[1]The first use of the term "envelope" to refer to tool wrapping, that we know of, was with respect to the ISTAR system [58].

are given on the command line), and return a simple status code. Notice this does not preclude interactive tools — even graphical user interface tools such as project schedulers and drawing programs — since the tool's own user interface appears on the user's display device when the envelope executes the tool. The user may then enter text or click menu items as desired; however, the granularity of access to objects/files from the environment's data repository is the entire tool invocation. In other words, the nature of current Black Box enveloping technology requires that the complete set of arguments from the repository is supplied to the tool at its invocation and that any results to be returned to the repository are gathered only when the tool terminates, so that the tool execution — what we call here an *activity* — is encapsulated within an individual task.

There are numerous tools whose natural and/or convenient use doesn't fit this description, but may be highly desirable to integrate into PCEs, including at least the following categories. Note these classes are not disjoint.

- Tools intended to support *incremental* request of parameters and/or return of (partial) results in the middle of their execution, such as multi-buffer text editors and interactive debuggers. Although such tools by definition allow submission of an arbitrary sequence of the user's choice of commands during their execution, when run in a stand-alone fashion, current enveloping technology does not permit the sequence of commands to be guided, automated or enforced by a task-oriented environment, and often even precludes retrieval of their parameters from the environment's data repository (e.g., if the process engine controls all access to the repository).

- *Interpretive* tools that maintain a complex in-memory state reflecting progress through a series of operations: Lisp applications, such as "Knowledge-Based Software Assistant" (KBSA) systems [39], are classic examples. Such tools may require severe start-up overhead and command substantial system resources (thus we refer to them as "heavy-weight"). We are particularly concerned with permitting different users to submit activities to the same tool execution instance, even when that tool was not designed to support multiple users. One of our goals is to extend a variety of single-user tools to (modest) multi-user operation.

- *Multi-User* tools, such as conventional database management systems that guarantee atomicity and serializability of separately transmitted but concurrently executing transactions. An important subclass is *Collaborative* tools (often referred to as computer-supported cooperative work — CSCW — or groupware), which abhor the conventional isolation model and directly support multiple users interacting with each other, such as WYSIWIS (what-you-see-is-what-I-see), IBIS decision support, Fagan-style document inspection, desktop video conferencing, etc. (see [129, 2] for more examples).

We introduce a *Multi-Tool Protocol* (MTP), where *Multi* refers to submission of *multiple* activities to the same executing tool instance and enabling of *multiple* users to interact with that same tool instance. Tool instances may operate for an arbitrary period of time, far beyond the length of an individual activity on behalf of an individual user; thus we refer to the executing tool instance as "persistent" with respect to the duration of the activities submitted under the MTP protocol. MTP also addresses *multiple* platforms: submitting tool invocations to machines other than were the user is logged in, e.g., when operating over a heterogeneous collection of workstations and server computers but executables are available for only a particular machine architecture or even only for a specific host; and *multiple* tool instances: managing a set of executing instances of a tool, e.g., when licensing limits the number of instances that can operate at the same time (common with commercial server licenses). MTP, as currently defined, treats tools in a Black Box manner. MTP has been implemented as part of the Oz process-centered environment.

Section 6.2 supplies brief background information on Oz. Section 6.3 introduces a tool modeling notation for specifying the category and special requirements of the tool; this notation extends Oz's previous facility, but could readily be adapted to other PCEs with some notion of tool declaration. Then we present our main work in Section 6.4, covering the general ideas, persistent tool sessions for four different categories of tools, an extension of the Oz client/server architecture for managing MTP tools (intended to be adaptable to other client/server or peer-to-peer architectures), the protocol for interaction between a process or task management engine and executing tool instances, and finally the structure of the tool wrappers themselves (we will use the terms "envelopes" and "wrappers" interchangeably throughout the paper). Then Section 6.5 describes four tool integration experiments, one for each of our categories. We discuss related work in Section 6.6. The paper concludes by summarizing our contributions and outlining future work.

6.2 Oz Background

Oz [27] is a process-centered environment framework. It represents both product (project artifacts) and process (workflow status) data using a home-grown object-oriented database management component, with a separate objectbase for each instantiated process. An object may contain zero or more file attributes, each typed as either `text` (ASCII) or `binary`. The value of a file attribute within an objectbase is a file system pathname into a "hidden" file system specific to that objectbase, not intended to be accessed except through Oz. Non-file attributes include the usual primitive values (strings, integers, etc.), containment of child objects, and references to arbitrary objects elsewhere in the same objectbase.

Oz's Shell Envelope Language (SEL) [86] is typical of current Black Box enveloping facilities, which typically involve some scripting language.[2] The process engineer (or environment builder) writes what are essentially UNIX *sh*, *csh* or *ksh* scripts, using added constructs that a translator expands into regular shell commands to handle the details of interfacing between the tool and the environment framework. An SEL envelope is associated with each task activity. After parameters have been bound and other preliminaries completed, Oz's process execution service directs that the named envelope be invoked on the arguments specified by the encapsulating task, including literals and/or object attributes. When the envelope terminates, it returns a status code and (optionally) result values to the process engine, at which point the pending task assigns the result values to objectbase attributes and performs various operations based on the envelope's status (typically indicating success versus failure).

The mechanisms described above are implemented within a client/server architecture, one server per instantiated process, as shown in Figure 6.1. Tool envelopes are forked by clients. The server sends envelope names and arguments to the client responsible for that activity, and then handles other clients in a first-come-first-served manner until the tool completes and the results returned by the client arrive at the front of the server's request queue.

The figure shows the main components of an Oz server: Inter-Process Communication (IPC) with its clients, Object Management System (OMS), Software Process Manager (PM), Transaction Manager (TM), and the "glue" that holds them together as well as performing multi-client scheduling (labelled Control). The clients have limited knowledge of object management (om) and process management (pm), and of course also include an interprocess communication component (ipc); the activity manager (am) is responsible for managing tool invocations. XView and Motif graphical user interfaces are supported, as well as a tty command line interface (not shown in figure). The various components are drawn as "jigsaw pieces" to denote numerous connections among components as opposed to, say, a purely layered architecture. See [31, 21] for additional details.

6.3 Tool Modeling

Assuming both SEL-like enveloping and a new MTP protocol are available, the process or other task-oriented execution service needs to specify which tools require which protocol. In principle, every tool could be invoked via the new MTP protocol, but we retained SEL for Oz (or the equivalent facility for some other system) as the default because we believe that MTP is complementary to

[2]SEL and many of the other Oz facilities mentioned in this paper were originally developed for our earlier system called MARVEL.

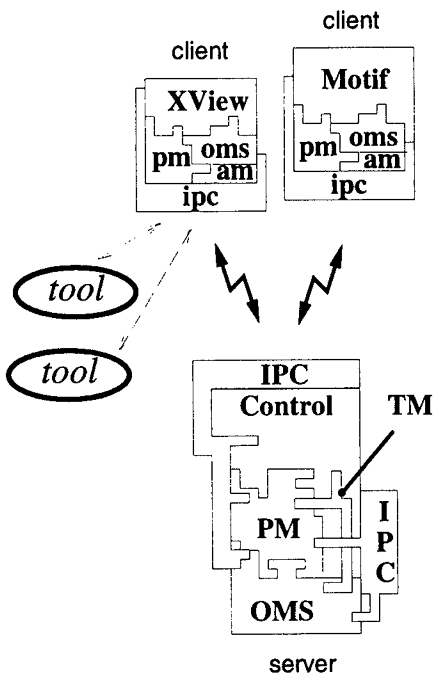


Figure 6.1: Original Oz architecture

SEL on a per-tool basis: together, they address with greater specificity the peculiarities of diverse families of applications, and the choice allows minimization of overhead balanced across a number of factors (see Section 6.4). In general, we believe an approach to integration based on *multiple enveloping protocols* is likely to achieve the greatest generality.

In the Oz implementation, the tool declaration notation has therefore been modified to include the new portion shown between square brackets ("[...]") in Figure 6.2, which is optional and may be omitted for SEL (some but not all of these fields are meaningful for SEL, as explained later, but defaults are assumed if they are not provided by the process engineer). Note each tool declaration is represented as a subclass of the built-in TOOL class; running tool execution instances are viewed as instances of these subclasses (although they are not currently reified in Oz's objectbase).

The new fields have the following meanings:

- `path`: indicates the pathname in the file system where the tool's envelope resides (or the tool's own binary executable, since an envelope is not always needed for tool initialization when using our MTP protocol, depending on the details of the tool). For example, an envelope might prompt the user for tool parameters not managed by the environment (such as a database volume).

- `host`: an Internet address, given when it is necessary to run the tool on a specific host because of some restriction (perhaps due to pragmatic licensing issues).

- `architecture`: used to indicate the machine architecture and/or operating system on which the tool (and its corresponding envelope) is expected to run. When the `host` is not specified, the system refers to the `architecture` specification and separate environment instance-specific configuration information, to determine a corresponding default machine on which the persistent tool (and its envelope) will be invoked.

- `instances`: This specifies the maximum number of copies of the tool that can execute at the same time (0 means there is no upper limit). Independent of licensing issues, this could be used to bound the system resources allocated to a heavy-weight tool in all its instantiations.

```
<tool-name> :: superclass TOOL;
  [ protocol    : (MTP, SEL) ;
    path        : <string> ;
    host        : <string> ;
    architecture: (sun4, ...) ;
    instances   : <integer> ;
    multi-flag  : (UNI_QUEUE, MULTI_QUEUE,
                   UNI_NO_QUEUE,
                   MULTI_NO_QUEUE) ;
  ]
   <activity-name> : string =
      "<envelope-name> <parameters locks>";
   <activity-name> : string =
      "<envelope-name> <parameters locks>";
   ...
end
```

Figure 6.2: Modified tool definition notation

- `multi-flag`: This determines the behavior of MTP in managing the interactions between multiple human users and a persistent tool instance. We distinguish among four categories of tools, with respect to their single-user versus multi-user and single-tasking versus multi-tasking capabilities, through the cross-product of two orthogonal dimensions:

  - **UNI** versus **MULTI**: MULTI (multi-user) indicates that the same instance of the program can be shared by several users, whereas UNI (single-user) allows only for isolated work of each user on his/her own executing instance of the tool;

  - **QUEUE** versus **NO_QUEUE**: where concurrent (overlapping) execution of multiple activities with respect to the same tool instance is supported for NO_QUEUE (multi-tasking) but not for QUEUE (single-tasking).

  It may seem counterintuitive to think of these dimensions as orthogonal. In the case of `MULTI_QUEUE`, i.e., multi-user and single-tasking, multiple activities on behalf of different users can share the same tool instance, but only one actually controls it and views the user interface at a time, in "floor-passing" fashion. For `UNI_NO_QUEUE`, i.e., single-user and multi-tasking, multiple activities can execute simultaneously in the same tool instance (perhaps in distinct "buffers" or other tool-specific contexts — the tool need not be implemented using multi-threading or parallel processing technology), but all must be on behalf of the same user. The four cross-product cases are explained by relatively generic examples in Section 6.4.1 and correspond to specific experiments elaborated in Section 6.5.

Each of the declarations following the brackets specifies the name of a activity together with the file name of an envelope, distinct from the one that started up the tool (if any). The activity-specific envelope is invoked whenever the corresponding activity is submitted to the persistent tool. There are likely to be several qualitatively different activities that can be performed using the same tool, so it is expected that multiple activity/envelope mappings would be listed in the tool declaration. If so, multiple instances of the same activity or several entirely different activities can be submitted to the same persistent tool execution. Formal parameters and locking information are also listed (transaction management is outside the scope of this paper, see [14, 105]). The envelope specified by the associated task handles the passing of arguments back and forth to/from the environment's repository as well as the details of interaction with a tool that is already running.

These declarations appear in identical form in SEL specifications, but in that case each envelope invokes a distinct tool instance to perform the activity (and envelopes may be grouped into the same tool declaration for abstraction reasons, without necessarily employing the same external application program). We made no changes at all to Oz's process definition facilities other than the tool declaration notation, and our approach is intended to be orthogonal to the environment framework's mechanisms for workflow definition and performance.

6.4 The Integration Protocol

We adopted what we call a *loose wrapping* approach, as opposed to the *tight wrapping* currently effected in Black Box enveloping schemes. The latter relies on complete encapsulation of all of the tool's actions inside a single envelope, whereas the former is instead based on control of the tool's behavior (from the viewpoint of the PCE), with the enveloping facility intervening only as the need arises during workflow execution and/or upon detection of some external event relevant to the environment. A typical example of the former is when the initiation of a process step (either automatically or through an environment command selected by a user) requires the tool to perform some work, and of the latter when a tool action saves some files that should be recorded in the environment's repository.

Control, as opposed to encapsulation, provides a means for long-lived and intermittent dialogue between external tools and the environment; meanwhile, the tools continue their execution effectively detached from the environment framework. Tight wrapping, on the other hand, governs all phases of a tool's execution, from the moment of invocation to termination; to perform multiple activities using the same tool, it must be explicitly and repeatedly instantiated (even if on behalf of the same user) each time an activity is assigned to the tool.

Our approach may be viewed as combining the advantages of conventional Black Box enveloping and event notification systems like Field and YEAST [198], where tools execute persistently but the server's concern is only for events of interest to other tools and there are no separate "environment commands" or "workflow" that control tools. The Forest extension of Field manages the propagation of event notifications among tools according to "policies" [79], analogous to Oz's process management services, and Provence is implemented on top of MARVEL [124, 108], the predecessor of Oz, but neither has any means for requiring satisfaction of task prerequisites. These systems also do not address one of our foremost requirements, to integrate multi-user tools, and few message buses are concerned with groupware or even support multiple users per bus. Buses internal to PCE frameworks such as ConversationBuilder [130] and ProcessWEAVER [67] are exceptions.

Once we established loose wrapping as the overall principle on which to base our design, we analyzed the major capabilities needed to implement our tool modeling facilities (described in the previous section). We divide these functions into two categories: those generally concerned with Black Box integration — i.e., the abilities to invoke and terminate an instance of a tool on demand, to parameterize that instance according to the corresponding process task, to transform objects from/to the environment's representation to/from that required by the tool, to support and display the I/O flow between the wrapped program and its user(s) — and those abilities especially necessary given the nature of the four tool categories of interest (i.e., the cross-product of UNI vs. MULTI and NO_QUEUE vs. QUEUE):

1. Limit the number of co-existing (executing) copies of a given tool according to the specifications set out in the tool's declaration, and to record and service previously unsatisfied requests as soon as possible;

2. Exploit the persistence of MTP-tools, in order to share a given instance among multiple users — possibly emulating partial multi-user capability for programs not usually employed for groupware;

3. Coordinate overlapping requests for access to an instance of a persistent tool from separate

```
OPEN-TOOL tool [session]>
    <MTP-activityA> <argumentsA> <session>
    <MTP-activityB> <argumentsB> <session>
    ...
CLOSE-TOOL <tool [session]>
```

Figure 6.3: Tool session template

users, to avoid deadlocks and starvation on the one hand, and of unintended concurrency of several activities for programs that do not support any form of multi-tasking on the other; and

4. Record results of intermediate steps of the tool's processing, during the execution of each single activity.

To fulfill these requirements, we have introduced several extensions to Oz's process management services. Analogous extensions could be made to other environment frameworks.

6.4.1 Tool Sessions

To encompass both serial and concurrent access to a tool instance, we introduce *sessions*, which define the life-span of a persistent tool. A session normally begins with an `OPEN-TOOL` command and ends with a `CLOSE-TOOL` command, as illustrated in Figure 6.3. A session's body is made up of a set of activities, denoted `MTP-activity` in the figure, determined dynamically as the users carry out their work within the environment. Note that although the activities are listed in sequence, they could potentially overlap (for NO_QUEUE tools).

`tool` could refer to any tool declared as MTP. The `session` identifier distinguishes among simultaneously executing instances of the same persistent tool, so that multiple users can choose to participate in a particular session opened by another user (for MULTI tools). Both arguments are selected from menus. Users can ask to join an existing session (if there are any) by clicking the corresponding automatically generated session identifier when issuing an `OPEN-TOOL` command, or request a new session as shown in Figure 6.4. The current implementation does not provide any support for access control, e.g., specifying which users are permitted to, or are required to, join a particular session. There is also no support for providing parameters for tool initialization from within the environment, which is less limiting than it sounds since the process steps that trigger incremental interaction with the tool usually provide arguments from the environment's repository.

Leaving a session is achieved with a `CLOSE-TOOL` command applied to a session where there are still other active users. In this case, the `CLOSE-TOOL` does not kill the tool instance, but only changes internal information about the association between the user and the session. Termination of the program follows the `CLOSE-TOOL` command of the last participant.

Besides setting the duration of a specific tool instance and providing a context for sharing an application, sessions are central in several other functions supported by our MTP protocol. For example, they implicitly operate on what we call the *Session Queue* of a tool. This feature allows us to satisfy the constraints posed by the `instances` field of a tool declaration, accordingly limiting the maximum number of copies of the program that can be active simultaneously. (Such a restriction could be violated due to tool instances executing completely outside the environment, resulting in tool invocation failures.) When `OPEN-TOOL` is issued, the system first checks whether the request is satisfiable given this constraint. If the limit has been hit, the request is not serviced, but is recorded in the Session Queue; when an already running session is terminated, the next queue entry is extracted and automatically initiated (the user is effectively notified when the user interface of the tool pops up on his/her workstation monitor).

Our design also allows for a special case where it is possible to use a persistent tool without being compelled to issue the `OPEN-TOOL` and `CLOSE-TOOL` commands every time, via an implicit

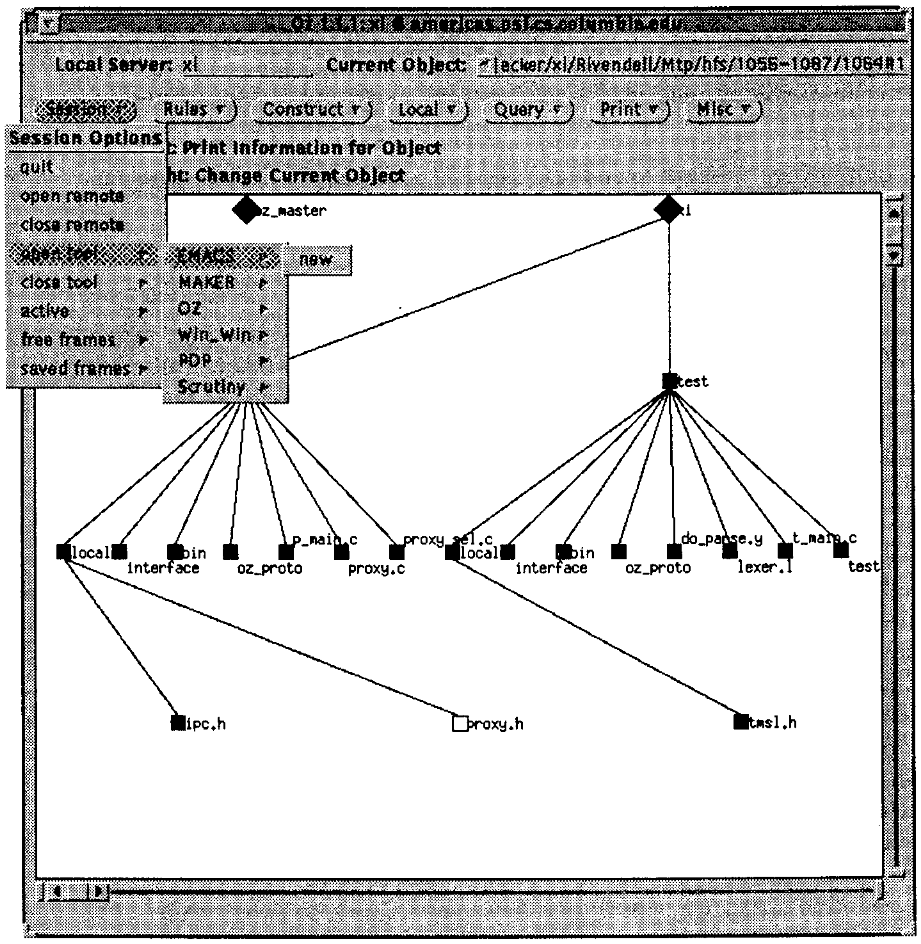


Figure 6.4: Oz MTP Interface

atomic session that consists of only a single activity. *Atomic* sessions are instituted by the system, transparently to the user, when a user intends to perform an activity associated with an MTP tool but has not previously opened or joined a session. In that case, an implicit OPEN-TOOL command is automatically executed and the new tool instance is marked as *atomic* by the environment, so that no other activities (or OPEN-TOOL/CLOSE-TOOL commands) can be directed to it. When the activity finishes, the tool is shut down automatically.

Our sessions idea leads to a number of questions on how different users could, practically, participate in the same session of a persistent tool, thus exploiting the same resources and the collected state of the executing tool. In our MTP design, we stressed the facets intended to accommodate in a natural way those applications that are inherently designed for collaboration, or — a more ambitious goal — to exploit in a multi-user context those tools that, even if not commonly employed in that manner, the environment builder considers adaptable to and promising for collaborative activities.

Our four categories of tools provide a flexible solution to these problems: the valid values of the `multi-flag` field within the tool modeling specifications represent and enforce in the protocol four working models, intended to cover as widely and as precisely as possible the behaviors and requirements of various classes of persistent tools.

UNI_QUEUE is the most basic category: with it, we intend to accommodate applications that are strictly single-user and that could not adequately support concurrent operations deriving from simultaneous MTP activities. Therefore each instance of such a tool is reserved exclusively to the user who requested it in the first place, via an OPEN-TOOL command, and the body of the session is made up of a simple sequence of activities that are never permitted to overlap.

The most significant difference between MTP's UNI_QUEUE and SEL is that multiple operations can be sent to the same copy of the tool, under the control of the process engine, by exploiting the newly introduced concept of *Activity Queues*: each UNI_QUEUE session is associated with an Activity Queue, which holds in first-come-first-served order the activities waiting to take control of the tool instance.

Consider, for example, a drawing program with a relatively long start-up time (e.g., it may load numerous fonts during initialization). Rather than force the user to wait several seconds to bring up the tool for each of the increasingly detailed data flow diagrams the process directs him/her to construct as part of a design document, the tool is invoked once and then this executing instance is reused for each separate diagram. This model assumes the tool provides interactive commands to load and store particular diagrams in the file system or a database, as most drawing programs do. Each activity begins by loading an existing diagram, indicating that a clean slate is needed, or simply expanding on the most recently loaded diagram, and ends with storing that diagram, with arbitrary tool-specific commands in between.

UNI_NO_QUEUE is intended to satisfy more complex integration requirements and to allow for more operational flexibility. Again, each tool instance is reserved for just one user, but the full exploitation of the inherent multi-tasking (or multi-context) capabilities of the tool is supported, by directing to the tool multiple simultaneous or overlapping activities.

One case is a multi-buffer text editor, where the user can easily switch among buffers with an interactive command; perhaps two or more buffers can be shown at the same time. A programmer might be part way through editing a particular source file when he/she realizes that it would be useful to cut and paste some code from another file, and modify the copied code, rather than type it in from scratch or call that other code as a subroutine. And while looking at this other source file, the programmer decides to make some changes to it, too, which may entail loading into the editor the header file(s) it imports, and so on. The process dictates certain obligations, such as recompilation, static analysis, and/or code inspection, for each edited file, perhaps somewhat different process segments depending on file type (source vs. header vs. documentation) and/or on whether the programmer is the "owner" of that file. Thus the editing of each file must be treated as a separate activity by the process, while at the same time it is useful to load the files into different buffers of a single executing instance of the editor rather than bring up a separate instance for each file.

If a tool is not inherently multi-user (as is the case for most current tools), but is declared

MULTI_QUEUE, only the most rudimentary form of sharing is possible: different users are allowed to join the same session, and therefore to access the same executing tool instance. But they must "take turns" (if they happen to issue requests that overlap in time): they are forced to wait in the Activity Queue until the previous activity is finished. Note that users whose requests are placed in the Activity Queue may still execute other process steps — or decide to abort and try again later (Oz's XView and Motif interfaces allow a user client to context-switch at will among in-progress process segments, and many other environments do likewise). Albeit limited, this form of sharing can be usefully exploited in various collaboration scenarios, for example, by multiple users committed to take care of different sequential stages of the same complex, long and composite process task, in which all must employ the same external program. One can then think of the MULTI_QUEUE tool as a semi-permanent environment service for these users.

Any interactive tool could, in principle, be supported by MTP as a MULTI_QUEUE tool. But it would not always be particularly desirable or useful to do so. Imagine declaring an electronic mail tool as MULTI_QUEUE. Then one user might read and respond to one incoming message, another user the next message, a third composes a new outgoing message, and so on. But such an activity sequence seems unlikely to be part of any practical software development process. Instead, MULTI_QUEUE is intended primarily for tools that build up a substantial in-memory state and that — under normal usage — support a sequence of activities that depend, at least in part, on the state constructed by previous activities and on the efforts of distinct human users (or user roles).

One example might be a Lisp-based application that generates natural language, say for a user manual, from a knowledge representation constructed during the requirements analysis and functional specification phases of the software process. A sequence of human-directed procedures are generally needed to turn the internal structure into prose appropriate for the end-user of the system under development. A software analyst might initiate the work, perhaps interleaved with activities performed by programming and/or quality assurance personnel, to be polished off by a technical writer and reviewed by a customer representative. Each user begins his/her activity where the last left off, with the tool's user interface automatically redirected among user display devices as another user takes over. The different user roles bring different kinds and levels of expertise to bear on producing the finished document. Note that while it is certainly possible to develop a knowledge-based assistant that saves its relevant state in the file system between steps, allowing separate invocations for each user, a given tool is not necessarily constructed that way. Further, even if such were available as an option (e.g., a Lisp image might be saved on disk), the heavy-weight start-up overhead might be best limited to a single invocation per process segment rather than once for each activity.

The MULTI_NO_QUEUE class was conceived to accommodate inherently multi-user systems, taking into account their architectural and functional peculiarities. MTP ensures in this case that every OPEN-TOOL command issued by some user in the context of the same session maps to the instantiation of a portion of the same multi-user system (e.g., a client in a client/server architecture), which is assigned to that user.

While MTP is in charge of directing users' process-determined activities to MULTI_NO_QUEUE tools, it is the intrinsic multi-user nature of these applications that defines whatever sharing and concurrency control policies are necessary to operate in the multi-user and possibly collaborative context. The transparency or visibility among user-controlled components with respect to their activities and data depends solely on the nature and the purpose of the tool, which may support collaboration (in a groupware application) or enforce isolation (in a conventional database management system). The integration protocol, per se, is not concerned with these issues.

An interesting MULTI_NO_QUEUE case is a process-centered environment, itself treated as a tool. The controlling PCE might specify the process at a relatively coarse granularity, e.g., coding and unit testing an individual module would be represented as a single task and integration testing of a subsystem as another. The controlled PCE (i.e., the "tool") might assist the users in carrying out the finer details of such tasks, e.g., editing, compiling, constructing test harnesses, and debugging would be separate steps triggered by the code-and-test task. (We have explored elsewhere the advantages of integrating higher level and lower level process definitions [127].) We assume here that the controlled PCE is itself designed and implemented as a multi-user system, e.g., following a

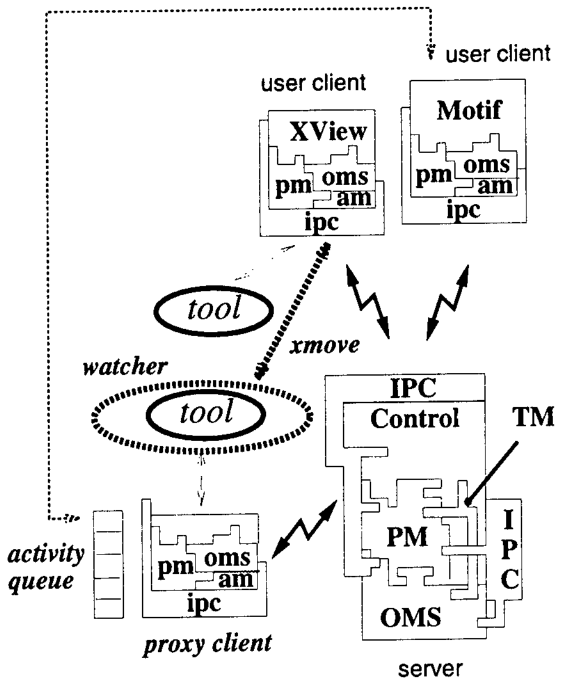


Figure 6.5: New Oz architecture

client/server architecture as in Oz, to allow teamwork within each coarse-grained step as determined by the finer-grained process. The two PCEs may or may not be distinct instances of the same system.

6.4.2 Architecture

The implementation architecture is necessarily specific to Oz, but we anticipate that a similar approach would apply to other multi-user process-centered environments. We divided Oz's clients into two categories, new *proxy clients* (or just proxies) and the original *user clients*.[3] Proxy clients introduce into the architecture a new kind of long-lived entity, with the role of spawning, managing, and achieving the integration of persistent tools. User clients are always associated with human users of the system, who invoke and exit them at will, and therefore they cannot be relied on to support the life cycle of a persistent tool instance. The Oz server persists indefinitely but provides process execution and object management services and most aspects of tool management discussed in this paper, but is intentionally not directly involved with tool invocation (in part for performance reasons, see [22]).

In our design, the session management commands (OPEN-TOOL and CLOSE-TOOL) are issued by user clients on demand by human users and executed by the appropriate proxy client, installed on the machine determined by the `host` or `architecture` data in the MTP TOOL declaration and, if both fields are null, then on the same machine where the Oz server is running. Subsequent activities submitted to the same tool may be initiated from a user client's user interface, but are delegated to the proxy client. The same proxy manages all persistent tools executing on the same host (with respect to activities managed by the same Oz server).

Proxy clients do not need to interact directly with any human operator, so no user interface is needed. However, they must manage the user I/O to/from persistent tools. This involves redirection of simple textual I/O between the tool and the user client, and more significantly the ability to display the tool's own graphical user interface (GUI) on the user's display. Most inherently multi-

[3]Proxy clients and user clients were initially referred to as Special Purpose Clients and General Purpose Clients, respectively [226].

user tools are able to dispatch private instances of their user interface to each user, but for other tools (e.g., originally single-user tools extended by MTP to a modest form of groupware) we exploited the public-domain *xmove* utility [215], which transfers the GUI of a tool across workstations and X terminals. Resetting the X Windows *DISPLAY* variable would be insufficient, since the GUI instance has to start on one display device for one user, then move to another for a second user, etc. without reinitializing the tool. (Note our implementation is inherently limited to those GUIs based on X Windows.)

Another job assigned to proxies is to spawn, manage, and communicate with auxiliary programs called *watchers*, each of which operates in the temporary directory for a tool instance and "notices" any files created or updated by a tool. These files are mapped to activity arguments according to a configuration file constructed by the envelope. The files can then be transferred back to the environment when the activity is completed.

The new proxy client, here supporting `MULTI_QUEUE` operation for a single persistent tool, is depicted in Figure 6.5. The internal composition of a proxy client is nearly the same as a user client, except there is no user interface and an additional component handles watchers, activity queues and other aspects of persistent tool management (the unlabelled piece of the proxy client in the figure). The same proxy client may manage multiple persistent tools, in which case there may be multiple activity queues — one for each `UNI_QUEUE` or `MULTI_QUEUE` tool.

Besides the capability for the same tool instance to handle multiple activities, another major difference between a SEL-like protocol and MTP's UNI cases, at least with respect to environment frameworks similar to the Oz architecture, is forking of the envelope and, indirectly, the tool by a proxy client — often not on the same machine as the user client — which could result in unnecessary communications overhead. MTP could easily be modified to default to a proxy on the same machine as the user client, and even some of the user and proxy client functions could be merged so that a separate proxy would not be needed when `host` and `architecture` specifications are not supplied and/or match the user client machine.

6.4.3 Envelope Execution

The most significant remaining issue that must be resolved to complete the design of our new protocol is the way in which the execution of envelopes is accomplished, in the manner of the ***loose wrapping*** concept. A typical MTP activity execution steps through the sequential phases listed below:

1. A **reservation** phase, in which a tool session is acquired on behalf of the activity and its associated user. This is carried out according to the session mechanism explained above.

2. An **initialization** phase, in which the objects/files from the environment are made available to the tool and any other parameterization functions are performed. We have employed for this purpose a standard envelope template, which accepts as its parameters: pathnames corresponding to file attributes in Oz's object management system; the path to a dedicated temporary directory that is created when the tool is started up and within which it normally operates; and some additional information used for internal housekeeping. The filename of this envelope is given by the tool declaration in its `envelope-name` field.

   The envelope is forked by the relevant proxy client, which sets up UNIX pipes for communication. The first job of the shell script is to copy the files into the tool's dedicated directory, thus making them visible to the tool; then any series of shell commands can be inserted, to perform whatever customization is necessary; finally, via the pipes, a sequence of text messages is sent to the proxy, to be displayed to the user in a pop-up window. These messages may include the values of primitive attributes from Oz's objectbase, and are intended to direct the loading of the files from the temporary directory into the memory of the application and otherwise instruct the user as to what to do. For example, the text presented in the window might indicate the command line or the mouse action that the user should enter to get started on the activity, although the details of performing the work are usually left to the user's own creativity and expertise.

Although we would have preferred a totally automatic loading procedure, as accomplished by SEL, that it is hardly possible given the inherent restrictions of the Black Box model: MTP tools are already running before the execution of any activity envelope, and therefore cannot be initialized according to the individual activities. Moreover, we cannot assume any special facilities on the part of the tool for simulating user input; redirecting "stdin" is generally insufficient for GUI tools. However, the envelope, via messages to the pop-up window, may still provide assistance and guidance to the users in a practical and convenient manner.

A Grey Box variant of MTP would overcome this drawback, since the tool's programmable facilities could act in collaboration with the envelope, producing and exchanging messages interpreted as directives to be executed by the tool. (Some Grey Box experiments have been conducted using SEL; see Section 6.5.2.) In the White Box case, this issue can usually be avoided entirely.

3. An **operation** phase, which includes free use of the tool with all its features, including manipulation of the loaded data. There is no restriction on the use of the tool, because it is accessed directly and not through any intermediary. The only requirement of the MTP protocol (that cannot, however, be enforced in the Black Box case) is that the execution must not be terminated through the tool's own internal command, menu button or procedure, but only via the environment's `CLOSE-TOOL` command. In addition, both MTP and SEL assume that users do not access the "hidden" file system sereptitiously, e.g., loading files into the tool outside the workflow, although there is nothing beyond an obscure organization and naming scheme (witness the filename the user is asked to type in Figure 6.9) to prevent them from doing so.

4. One or more **data recording** phases may interleave with other actions, whenever the user saves temporary results of the work he/she is performing (the tool updates the copies of the files kept in its own temporary directory, and not those internal to the environment). Such events are monitored by the proxy client's watchers. A table of updated files is maintained in the proxy and used in the next phase.

5. The **conclusion** of the activity, at which point control of the tool is released (with respect to this activity). The user is required to designate the activity's completion as either a `success` or a `failure`, via corresponding buttons in an MTP-specific extension to Oz's activity management window (see Figure 6.9). The data resulting from the execution is stored back in the environment only if the user considers the activity successfully completed.

   SEL expects the envelope to automatically capture the return code of the tool after the user decides to close it, but in MTP the tool remains indefinitely active; therefore the only means of ending an individual activity is to let the user decide when his/her work is finished and to provide a way to communicate this fact (and how to handle the results) to the envelope. Other differences are that SEL file updates are permanent, regardless of the success or failure status: actually, SEL may return any value in a range determined by the encapsulating task, each of which will result in different obligations following that task. A similar facility could be added to MTP.

6.4.4 Wrapper Structure

Envelopes provide a very flexible approach to tool integration. Consisting of either standard scripts in some scripting language (as we have employed for MTP), or augmented variants of the scripting languages that provide primitives to handle interfacing to the environment and its data repository (as in SEL) — or possibly even written in a conventional programming language, wrappers offer programmable facilities that can handle the different needs and idiosyncratic properties of a wide range of external applications in a convenient and uniform way.

MTP uses two kinds of envelopes: the first is executed in response to the `OPEN-TOOL` command, whereas the second operates at the granularity of the individual activity. The latter is concerned

```
#!/bin/sh
#initialize variables
SERVER_PID=-1
CLIENT_PID=-1
# look if already hooked to the environment directory
FOUND=`find . -name linkfile -print`
# if environment directory is not found
if [ "x$FOUND" = "x" ]  #no oz_server active
then
        #The OZ environment directory is not set up
        #The shell script exit with -1
        echo "The OZ environment directory is not set up properly" \
            >> /tmp/SPC.log
        exit -1;
else
        #Change to the OZ environment directory
        cd linkfile
        #test whether there is a server running
        SERVER_PORT=`find . -name .server_port -print`
        if [ "X$SERVER_PORT" = "X" ]   #No server is running
        then
                #bring up the oz server
                /u/bleecker/xi/bin/oz_server &
                SERVER_PID=$!
                #Record the server process id
                echo $SERVER_PID>.server_pid
                #Record the number of client run on the server
                echo "0">.client
                sleep 5
        fi
        #start up the client
        /u/bleecker/xi/bin/gpc -x
        CLIENT_PID=$!
        #increase the number of clients
        read CLIENT_NUMBER <.client
        CLIENT_NUMBER=`expr $CLIENT_NUMBER + 1`
        echo $CLIENT_NUMBER >.client
fi
CURR_DIR=`pwd`
# trap a request to kill this OZ component and
# invoke close_oz_script to take care of this task.
trap '/u/bleecker/xi/Rivendell/Mtp/mtp/close_oz_script \
    $CURR_DIR $CLIENT_PID; exit 1' 2
wait
```

Figure 6.6: Example initialization script for a multi-user client/server tool

```
#!/bin/sh
# $1 tool_directory
# $2 oz client process id
echo "Close the client and server!\n" > &2
read CLIENT_NUMBER < .client
if [ $CLIENT_NUMBER != 1 ]
then
        CLIENT_NUMBER=‘expr $CLIENT_NUMBER - 1‘
        echo $CLIENT_NUMBER > .client
        kill -9 $2
else
        read SERVER_PID < $1/.server_pid
        kill -9 $2
        kill -2 $SERVER_PID
        #take care of the garbage
        rm $1/.client
        rm $1/.server_pid
fi
```

Figure 6.7: Example termination script for a multi-user client/server tool

mainly with preparing and loading the data that must be processed by the program during the associated activity; the former is used to perform customization of the tool, in order to present it to the user(s) in the correct state, in relation to the characteristics of the system and the work model indicated by the `multi-flag` specification. This kind of customization script is usually very simple — no more than a few lines of straightforward shell commands — but sometimes may be quite complicated, accounting for complex interactions with the environment through watchers, and sometimes even for the invocation of other auxiliary (usually simpler) scripts that perform supplementary bookkeeping or actions in response to particular states of the application. An example of an intermediate case is shown in Figures 6.6 and 6.7; note the latter shows the contents of the auxiliary `close_oz_script` invoked by the former.

In the case of the Oz implementation, the envelope writer must be a relatively skilled shell programmer with some knowledge of the purpose and the functions of the wrapping protocol to be able to easily set up the scripts. The burden might be lowered somewhat if MTP were to extend the scripting language with special-purpose primitives, perhaps somewhat different sets to accommodate each of the four work models. However, the experience gained with SEL shows that even with such primitives the scripts are not exactly trivial, since the intrinsic specificity of the application programs necessitates ad hoc treatment for each case.

Language extension would be useful mainly to abstract and parameterize those operations that must be carried out in a repetitive manner for any application; this seems more plausible with the data interface between the tool and the environment, rather than with the adaptation of their reciprocal behavior. Consider the example shown in Figure 6.8: some of the shell commands, those marked with the comment `# always`, must always be present in any MTP activity-related envelope; others, indicated by the comments that contain the words `FILE parameters`, are needed to handle certain types of incoming data, and are similar but not identical in all the envelopes. These two sets of commands together contribute to preparing the data involved in the activity.

The other shell commands, marked by the `# tool-dependent` comments, are concerned with operating the tool towards the goal of the task at hand. It is clear that in the general case the size and the complexity of this last set is dependent on the wrapped application, of the supported work model and, especially if a lot of direct interaction with the user is necessary, of the activity to be performed. In contrast, the former two sets are relatively independent of all these factors; hence it

```
#!/bin/ksh
#input parameters:
# $1 tool dir.          <----- MTP additional parameter
# $2 C file             <----- NOTE: FILE parameter
# $3 compile status     <----- Literal
# $4 compile log file   <----- NOTE: FILE parameter
# $5 C file proto       <----- For later extension to match
# $6 local project tag  <----- SEL editor envelope functionality
# $7 EnDoFAtTrSEt       <----- marks end of arguments from process
# $8 task identifier    <----- MTP additional parameter
# $9 client identifier  <----- MTP additional parameter

LOGFILE="/tmp/ForkLog"                    # debugging code
echo "start up enveloper" >>$LOGFILE     # debugging code
cp $2 $4 $1                     # copy all FILE parameters into the tool dir.
CFile=`basename $2`             # for all FILE parameters
CompileFile=`basename $4`                 # for all FILE parameters
CPath=`echo $1/$CFile`                    # for all FILE parameters
CompilePath=`echo $1/$CompileFile`        # for all FILE parameters
F_LIST_DUMMY=$1/filelist_tmp                      # always
F_LIST=$1/filetable                               # always
touch $F_LIST_DUMMY                               # always
echo $9 $8 $CFile $2 >> $F_LIST_DUMMY     # for all FILE parameters
echo $9 $8 $CompileFile $4 >> $F_LIST_DUMMY
                                          # for all FILE parameters
echo $F_LIST_DUMMY >>$LOGFILE             # debugging code
FOUND=`find $1 -name filetable -print`            # always
if [ "x$FOUND" = "x" ]                            # always
then                                              # always
        mv $F_LIST_DUMMY $F_LIST                  # always
else                                              # always
        F_LIST_CAT=$1/merge_list                  # always
        cat $F_LIST_DUMMY $F_LIST > $F_LIST_CAT # always
        rm $F_LIST_DUMMY                          # always
        mv $F_LIST_CAT $F_LIST                    # always
fi                                                # always
echo \#***\#TYPE: CTRL-xf $CPath
                                # tool-dependent : load code file
if [ $3 = "NotCompiled" ]       # tool-dependent
then                            # tool-dependent
        echo \#***\#TYPE: CTRL-x 2
                                # tool-dependent : display new buffer
        echo \#***\#TYPE: CTRL-xf $CompilePath
                                # tool-dependent : load compiler logfile
fi                              # tool-dependent
```

Figure 6.8: Example activity script for a multi-tasking tool

would be easier to invent scripting-language extension facilities to express them.

However, it would also be possible (and desirable) to define some ad hoc constructs for use in those tool-dependent statements that communicate to the user the actions that he/she should perform, e.g., to carry out the loading of activity arguments into the tool instance, during the initialization portion of an MTP activity. In Figure 6.8 these messages are implemented simply with `echo` commands prefixed by a common string (`#****#`); the output is redirected through pipes maintained between the envelope and the proxy client that initiated it, and the proxy is in charge of displaying the messages to the user in a pop-up window. One could certainly imagine more sophisticated facilities for guiding the user.

6.5 Tool Integration Examples

To test the facilities described in the previous sections, we have used several available in-house applications and off-the-shelf tools. The purpose of these tests was to gain confidence in the viability of the new MTP protocol, and in particular to challenge its ability to accommodate a wide range of variability in the nature of the wrapped applications.

Therefore, we have tried to define the degree of integration that can be reached and to identify limitations (either based on the characteristics of the tool category under examination, or specifically to the adequacy of our support to the single cases) or unresolved problems we need to address during future development. The applications we used as examples were:

- `idraw` as a `UNI_QUEUE` tool, where activities are queued for one-at-a-time execution (the same user may submit activities from multiple Oz clients, and the user interface is transferred among workstation monitors as needed);
- `emacs` as a `UNI_NO_QUEUE` tool where steps are not queued but may overlap (typically on a single monitor);
- A Lisp-based natural language processing system called *FUF* as a `MULTI_QUEUE` tool, where steps are queued for one-at-at-time execution (and the user interface is transferred among users participating in the same session as needed); and
- Oz itself as a `MULTI_NO_QUEUE` tool (that supplies its own clients for multiple users).

6.5.1 UNI_QUEUE: idraw

idraw [228] is a popular public-domain drawing tool, commonly used to develop pictures and diagrams stored in a postscript form. It provides an intuitive graphical user interface employing a well-known paradigm based on mouse movement and menu selection to operate on a virtual canvas shown within an X window. *idraw* is intended to be single-user; although it supports multiple buffers, we ignore that feature here, and treat the system as if it were necessary to save the current document before loading a different one. This limited use of *idraw* serves as an example of the category of programs where such restrictions are inherent. From our point of view, *idraw* presents some additional features of interest since it fulfills our definition of heavy-weight tool: there is a relatively long initialization time following its invocation.[4]

In our experiment, we employed a distinct activity, parameterized by a file attribute from Oz's objectbase, to construct a complete diagram or to allowing editing of an existing diagram stored in that file, with the details of the drawing left to the creativity and expertise of the user. That is, a activity's envelope sends a message to be displayed in a pop-up window, telling the user to load a file with a particular pathname, and briefly instructs the user regarding the purpose of the drawing to be constructed for that file. The user is responsible for using *idraw*'s normal command to later save that file, prior to announcing the conclusion of the activity. This accounts for a simple interaction model

[4] *idraw* takes about 15 elapsed seconds to start-up on a Sun SparcStation 10 workstation.

that is common practice in the use of such kind of tools; however, it would alternatively be plausible to invent activities and corresponding envelopes to operate at a much finer level of granularity, for example, "select the line icon and insert a vertical line two inches to the left of the triangle", but we doubt this would be useful (except perhaps as part of a tutorial in the use of a system devoted to the management of graphic documents).

The construction of the corresponding wrapper, and of wrappers for most `UNI_QUEUE` applications, is actually very simple: the only tool-dependent statements are aimed at instructing the user on how to load the input file and (optionally) on what he/she must do with it.

A few words are in order regarding our intentionally restrictive use of *idraw*: we had some trouble finding a good candidate for the most basic `UNI_QUEUE` category, among the interactive tools we had on hand for testing (SEL seems adequate and completely satisfactory for non-interactive tools, such as compilers, that must be restarted for each new set of arguments anyway); *idraw* on the other hand seemed to have many of the properties that we were looking for in a `UNI_QUEUE` candidate. However, we recognize that it would normally be deemed `UNI_NO_QUEUE`, because of its intrinsic multi-buffering capability (see Section 6.5.2). Further, one could imagine employing *idraw* in a multi-user context, where one user starts a picture and others add to and finish it, analogous to the work model in Section 6.5.3, in which case *idraw* could even be designated `MULTI_QUEUE`.

Given all of the above, one may have the impression that perhaps the `UNI_QUEUE` category is not really necessary. However, we expect that environment builders will discover cases where they intend a tool to be used in a certain restricted way within the workflow, and enforcement of `UNI_QUEUE` would prove useful.

In general, `UNI_QUEUE` appears suitable to deal with those applications that do not present any multi-tasking capability and do not seem particularly adaptable to multiple users, but are most conveniently handled as persistent tools. The main advantages of persistence for this class of tools, and the most valuable improvements introduced by MTP's loose wrapping compared to tight wrapping as in SEL, is the reduction of start-up overhead (since the tool need be invoked only once) and the user can run ordered sequences of activities on the same instance of the program without losing its internal state.

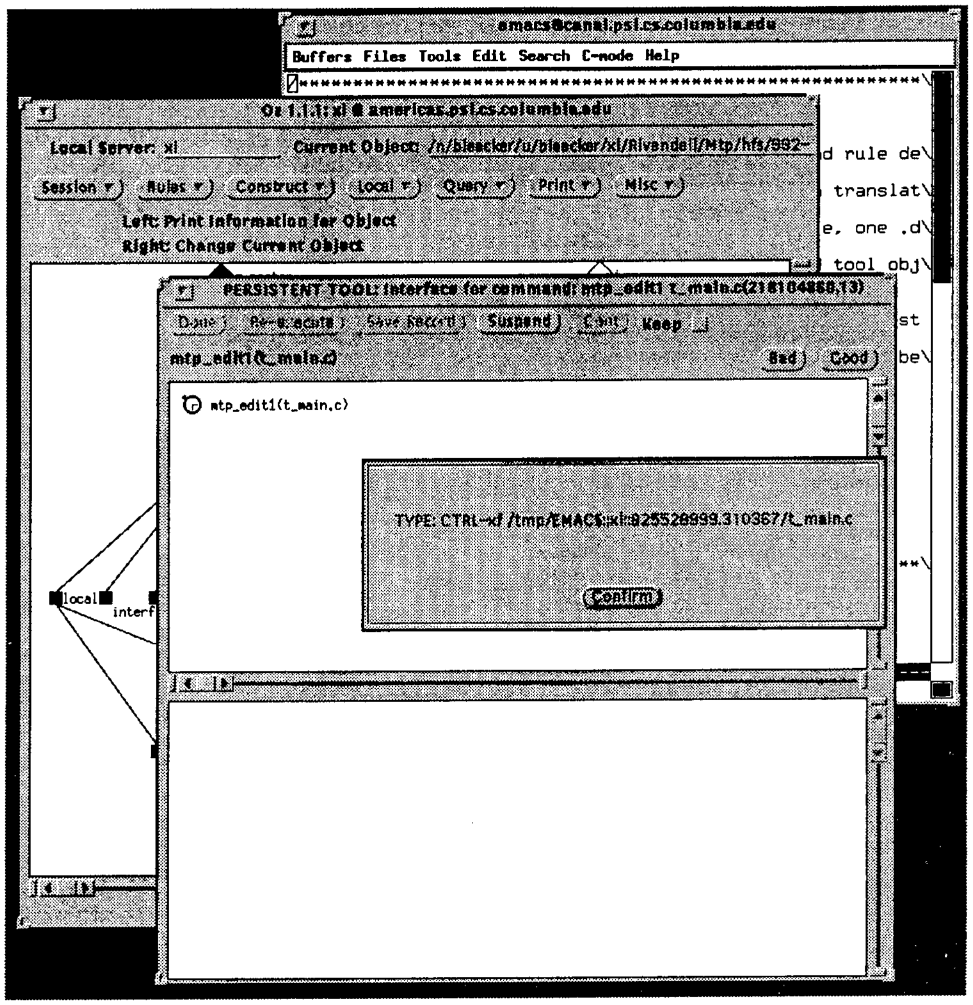
6.5.2 UNI_NO_QUEUE: emacs

emacs [216] is one of the most readily available and widely used text editors; its sophisticated functionality and features make it a very useful tool, which nearly reaches in itself the status of a single-user programming environment. All of its commands are expressed with sequences of keystrokes, augmented with mouse pointing and selection; its latest versions also support menu selection, at least for its main features. One of the most useful properties of *emacs*, and one of the most important for us with respect to this discussion, is its buffering capability. This enables the user to operate simultaneously on multiple files, keeping several buffers in the background and switching among them on command. Coupled with the ability to split the display and hence show more than one of the buffers, this feature is of great use to perform complex and incremental editing sessions that involve as many different data sets as needed.

Many users would prefer to use *emacs* in the natural fashion available outside a process-centered or otherwise task-oriented environment framework, which is to create and kill buffers, load and save files, and cut and paste among buffers/files, as the urge arises during perhaps very long work sessions.[5] *emacs* demonstrates the most obvious limitation of conventional Black Box wrappers — that is, all arguments must be supplied on the command line at tool start-up — in which some peculiarities of the application do not fit well with the protocol's design and are left unsupported, but it is nevertheless possible to integrate the program in some form.

MTP's `UNI_NO_QUEUE` class allows for overlapping multiple activities that involve loading various buffers of the same executing *emacs* instance with the desired files for the user's editing sessions. MTP then employs watchers to allow mapping of each modified file to the corresponding activity

[5]The second author has been known to keep the same *emacs* instance running for months, obviously persisting over numerous and often unrelated tasks.


emacs@canal.psl.cs.columbia.edu
Buffers Files Tools Edit Search C-mode Help
Oz 1.1.1: xl @ americas.psl.cs.columbia.edu
Local Server: xl
Current Object: /n/bleecker/u/bleecker/xl/Rivendell/Mtp/hfs/992-
Session
Rules
Construct
Local
Query
Print
Misc
Left: Print Information for Object
Right: Change Current Object
PERSISTENT TOOL: interface for command: mtp_edit1 t_main.c(218104868,13)
Done
Re-execute
Save Record
Suspend
Quit
Keep
mtp_edit1(t_main.c)
Bad
Good
Confirm
local
interf

mtp_edit1(t_main.c)

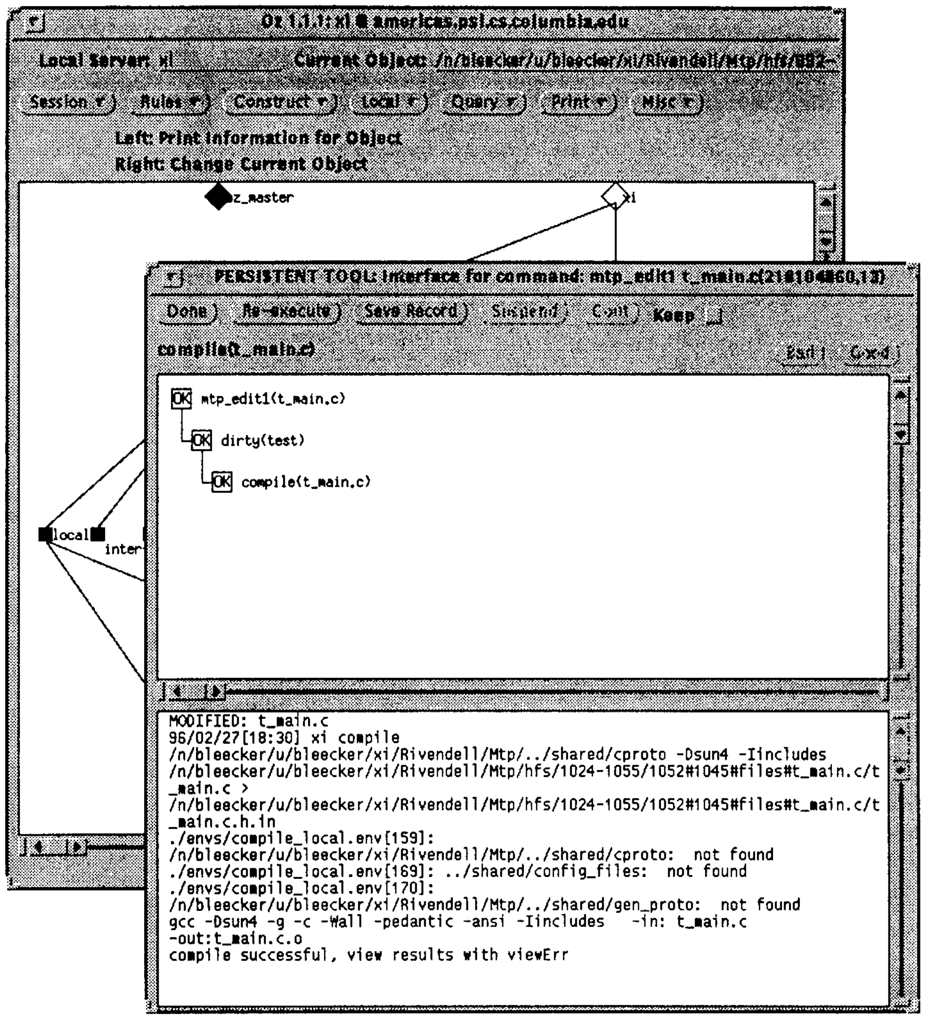
TYPE: CTRL-x f /tmp/EMACS:xl:825528999.310367/t_main.c

Figure 6.9: MTP Activity Initiation

and hence discriminates what file attributes must accordingly be modified inside the environment at the end of the activity. The use of a pop-up window during the initialization phase of each activity, and extensions to the standard activity window to indicate completion, effectively isolates the overlapping activities, in the sense that their data flow and status with respect to the on-going process are independent.

In our experiment, we employed individual activities, parameterized by file attributes, to edit programming language or documentation files; the details of the programming or writing were the concern of the user. That is, an activity's envelope would display a message on a pop-up window telling the user to load the file with a given pathname, as shown in Figure 6.9, and perhaps briefly explain to the user the purpose of the code or prose in that file (not shown in the figure). Rather than simply asking the user to edit, the envelope might instead request the user to repair the syntax errors found during the last compilation — by sending a file containing those error messages to another buffer as part of the same activity.

The complete script of an *emacs* wrapper of this kind is shown in Figure 6.8; it performs the loading of a C source file together with the results of the last compiler run, if unsuccessful, to display the generated error messages. Again, the user must give *emacs*'s normal command to save the source file. He/she may choose to indicate that the completion of the activity has been successful, by committing changes to the environment's repository via the **Good** (success) button in Oz's activity


Oz 1.1.1: xi @ americas.psl.cs.columbia.edu
Local Server: xi
Current Object: /n/bleecker/u/bleecker/xi/Rivendell/Mtp/hfs/882-
Session
Rules
Construct
Local
Query
Print
Misc
Left: Print Information for Object
Right: Change Current Object
z_master
xi
local
inter
PERSISTENT TOOL: Interface for command: mtp_edit1 t_main.c(2181048860,13)
Done
Re-execute
Save Record
Suspend
Cont
Keep
compile(t_main.c)
Bad
Good
OK
mtp_edit1(t_main.c)
OK
dirty(test)
OK
compile(t_main.c)

```
MODIFIED: t_main.c
96/02/27[18:30] xi compile
/n/bleecker/u/bleecker/xi/Rivendell/Mtp/../shared/cproto -Dsun4 -Iincludes
/n/bleecker/u/bleecker/xi/Rivendell/Mtp/hfs/1024-1055/1052#1045#files#t_main.c/t
_main.c >
/n/bleecker/u/bleecker/xi/Rivendell/Mtp/hfs/1024-1055/1052#1045#files#t_main.c/t
_main.c.h.in
./envs/compile_local.env[159]:
/n/bleecker/u/bleecker/xi/Rivendell/Mtp/../shared/cproto:  not found
./envs/compile_local.env[169]: ../shared/config_files:  not found
./envs/compile_local.env[170]:
/n/bleecker/u/bleecker/xi/Rivendell/Mtp/../shared/gen_proto:  not found
gcc -Dsun4 -g -c -Wall -pedantic -ansi -Iincludes   -in: t_main.c
-out:t_main.c.o
compile successful, view results with viewErr
```

Figure 6.10: MTP Activity Completion

window. Then the workflow may automatically continue to other tasks, as illustrated in Figure 6.10, where MTP and SEL activities may be arbitrarily intermingled in a single process fragment. Or the user decides not to save his/her work, by selecting the **Bad** button (failure), which has the effect of withdrawing whatever intermediate saves were performed during the work and noticed by the watchers. As with *idraw*, we did not consider finer-grained activities such as "add a new floating point variable to function f and initialize it to *pi*", but the implementation supports them.

A previous attempt to extend Oz's enveloping mechanism had focused on *emacs* as a test case, and tried to resolve the problems posed by the desired incremental data exchange with the environment. This previous attempt exploited a facility not provided by most tools: an extension language. *emacs*' extension language, called *E-Lisp*, allows users to define their own new functions and commands, and thus customize *emacs* to their applications.

Ad hoc *E-Lisp* functions were coupled with an augmented version of SEL, to effect a Grey Box integration, where the environment could perform loading of additional files into the same *emacs* instance at any time and discern which files had been updated. No special effort was required by the user, in contrast to the attention he/she must pay to MTP's pop-up window. This was achieved using one wrapper for the entire session, which dealt with addition of new buffers as new activities were submitted, rather than using a separate wrapper per activity. There was a major drawback to this approach, however: only one final status result could be returned to the environment, when *emacs* and its wrapper terminated, and all files were effectively recorded into the environment's repository at this same moment. In other words, it was not possible for the process to treat separately the different sets of data acquired throughout the work session — a central feature of MTP.

Later during the development of MTP, we looked at *E-Lisp* again to pursue Grey Box integration. Ad hoc *E-lisp* functions implemented a direct interface between *emacs* and the watcher utility, and also completely automated the initialization phase of the activities. The conclusion phase, particularly the choice of the `success` or `failure` return status for the separate activities run on the same instance of *emacs*, is still an explicit responsibility of the user even under this paradigm.

In general, `UNI_NO_QUEUE` appears appropriate for tools with some internal multi-tasking, multi-buffer or multi-context capability, but still not particularly useful or desirable for multi-user access. The main advantage of persistence for this class of tools is that the user can run partially ordered activities on the same instance of the program, without losing its intermediate state information, and possibly allowing for sharing or splicing (cut-and-paste) of intermediate results. Cut-and-paste can be intentionally directed <u>among</u> activities directed by the process, or even <u>within</u> a single activity that simultaneously presents multiple file arguments to the tool, in either case with the envelope's messages to the pop-up window instructing the user what to do. Note there is no means for preventing, from the environment, user-initiated cut-and-paste once the tool is designated as `UNI_NO_QUEUE`.

6.5.3 MULTI_QUEUE: FUF

FUF is a sophisticated unification-based tool running on top of Lisp and is used, among other things, in natural language processing research for the generation of sentences from corresponding syntactic data structures [59]. It defines hierarchical procedures that apply in sequence one or more separate layers of unification rules to its input structures — as well as to the new structures produced by each step of the procedure — in order to obtain as output all the valid surface forms, under the constraints posed by the language rules. *FUF* is a typical Lisp-based interpreted application, in that it that supports various kinds of interactive tracing facilities and has the option to test and execute various data and program files, by loading and swapping them on the fly. As with most interpretive tools, it maintains sufficient information in memory to reflect the progress of its elaboration through the series of commands issued to it since start-up. Moreover, like many query systems constructed on top of Lisp, there is a long start-up time and it engages a considerable amount of system resources (notably main memory and swap space) and thus qualifies as a heavy-weight tool.

One of the main reasons for this choice as our exemplar `MULTI_QUEUE` tool is that it is easy to imagine a scenario in which, in order to process some data with *FUF*, multiple unification procedures are needed, each of which is the responsibility of a different member of a development group. Our paradigm could facilitate the testing and execution of the various phases of the project through a (modest) form of groupware: sequentially, each developer would load into *FUF* its own program, run it on the appropriate data and refine it as much as needed, and produce at the end an output that is also the input for the next step, also leaving the system in the correct state to begin the following activity. MTP moves the user interface among the users as they take their turns. The final outcome of the overall workflow would be produced by a single instance of the system and as the result of the collaboration of several users. Analogous collaborative work models could be applied to other programs, which outside the MTP framework could not be employed in this way. We have recently used the commercial *FrameMaker* word processing system in `MULTI_QUEUE` style; although it supports multiple buffers, it does not provide machinery for multiple users and thus GUI movement support is needed.

The envelopes we devised for this case study are devoted to loading within the memory of *FUF* a specific unification program, and to handle the correct system configuration for it, by asking the user to type the appropriate Lisp commands. The user might know little, if anything, about the configuration issues involved: he/she needs only to follow the instructions appearing in a pop-up window, since each envelope is specialized towards a separate portion of the group work. After this initial customization, the user is left completely free to query *FUF* and interact with it in the typical fashion of Lisp-based interpretive applications. Any files produced as result of these operations may be imported into the objectbase when the success choice ends the activity, as described above.

From a general point of view, the MULTI_QUEUE category allows the reuse of single instances of such computationally expensive programs throughout a series of activities. Another important point in favor of supporting this class is that the information retained in the tool's memory space (and not necessarily persistently on disk) represents both the current state of the system and the history of its past performance, and is generally necessary for generating the answer to new queries. This makes even more valuable the ability of the MULTI_QUEUE work model to support applications with long-duration work sessions that go beyond any individual process step, and to ensure common access to them to any set of users.

The most relevant consequence of the creation of this category is indeed that, by exploiting Activity Queues and the *xmove* facility that achieves passing of control over the user interface among users involved in a session, it allows us not only to conveniently integrate a vast and peculiar family of tools, but also to actually modify at the same time their intrinsic single-user nature and extend their use along the serial groupware lines described above. We consider this as one of the most interesting and meaningful results of this work.

6.5.4 MULTI_NO_QUEUE: Oz

We decided to use Oz itself as a testbench for the MULTI_NO_QUEUE category. The main reasons for this choice were the familiarity we have with Oz as a complete multi-user system and the in-house availability of the application in a ready-to-run state. Oz, as a typical client/server system (and unlike most applications based on peer-to-peer architectures), poses, in the most general case, the problem of treating differently the OPEN-TOOL command initiating a session, when it is necessary to start-up both the tool's server and a client, from those subsequently issued to join the session, which obtain further copies of only a client. Conversely, the last CLOSE-TOOL command in a session must deal with shutting down the tool's server. Moreover, since one can optionally employ a daemon that automatically starts up the Oz server with the first client and automatically shuts it down when the last client exits, Oz can also be used to simulate the behavior of non-hierarchical architectures, which do not need special treatment for the activation of its first and termination of its last components.

The intrinsic difficulties of dealing with these issues were solved in the context of the envelope indicated by the path field of the tool declaration and invoked by the OPEN-TOOL command. The designated envelope is invoked exactly once per session for all other categories of tools, but in the case of MULTI_NO_QUEUE is invoked separately for each user who joins the session — and thus must be able to, internally, distinguish its first from its subsequent invocations with respect to the same persistent tool. Oz's initialization envelope is shown in Figure 6.6; this envelope handles the shut-down of Oz's server by invoking the auxiliary script given in Figure 6.7. MTP, with its MULTI_NO_QUEUE class, is therefore able to support a generic multi-user tool, by forking and providing copies of the program to every participant in a session, as required by its structure.

MTP could easily be extended to allow for two distinct initialization envelopes in the MULTI_NO_QUEUE case, or in all cases — so that the first user to join a session and all subsequent users may be treated differently (of course the two scripts may be identical if no distinction is needed for the particular tool). Similarly, MTP could be extended to handle yet another separate envelope triggered by the CLOSE-TOOL command, or a pair of envelopes distinguishing between the last user to leave a session and all previously exiting users.

During our experiment with Oz, we devised MTP activities that perform operations within an in-progress workflow (the process state as well as the product data is persistent across sessions as well as tasks and activities within a session). Some wrappers instruct the user, with the usual pop-up messages, on how to use Oz's GUI to browse the objectbase, inspect the process definition task set, etc.; this could be useful for training new users. More significantly, it is also simple to ask users to initiate specific Oz tasks, or sequences of tasks. Alternatively, the MTP activity might simply instruct the user(s) as to *what* is to be accomplished, and leave it to the user(s) to determine *how* best to achieve that goal within the process supported by the MTP-invoked Oz instance (not to be confused with the MTP-invoking Oz instance)

This raises the possibility of an Oz meta-process that controls one (or more) Oz process(es), effecting a form of hierarchical workflow system. This could potentially address a certain limitation of Oz as a PCE, namely that relationships among tasks within a process are formed only with respect to satisfying local constraints, the task prerequisites and obligations, and there is no global topology or "grand view" [127]. However, that grand view could feasibly be defined by the meta-process, by directing the workflow among abstract or at least aggregate tasks, while each MTP-invoked process itself directs only the workflow among concrete, perhaps primitive tasks, effectively filling in the details left out of the meta-process. The meta-process hierarchy could be elaborated to arbitrarily many levels, not just two. Further discussion of this idea is outside the scope of this paper.

There are some important differences between the integrations of collaborative tools, like Oz, and non-collaborative tools, which must be taken into account when considering the capabilities of the `MULTI_NO_QUEUE` work model. In the non-collaborative case, by definition each user is intended to be isolated from the rest and data access conflicts among overlapping argument sets are sporadic. In the case of data from the environment's repository, conflicts may be resolved before the arguments are passed to the tool by some concurrency control mechanism provided by the PCE; Oz, by default, implements conventional atomic and serializable transactions composed of individual or multi-step tasks [105]. When an external repository specific to the tool is employed (e.g., a database volume), the tool is assumed to have its own intrinsic concurrency control facilities.

In the collaborative case the issue of shared data becomes more problematic, even though most of the multi-user machinery is necessarily offered by the wrapped tool itself. A simple example is that of a multi-user editor [51] invoked in the context of a groupware activity: the program itself permits and is able to deal with concurrent modification of its internal data, but from the viewpoint of environment's data repository it is necessary to support a concurrency control policy that allows multiple writers of the object containing the edited file attribute(s); this is achieved in Oz by defining and loading application-specific concurrency control policies, written in a notation [102] that permits definition of extended transaction models including "cooperative transactions" [118]. Concurrency control, per se, is not in the strictest sense part of the wrapping facility, but is nevertheless essential in order to fully integrate this class of tools. Further discussion of this topic is outside the scope of this paper.

6.6 Related Work

As we pointed out in the previous sections, tool integration is of central importance to every effort to build efficient and practical software engineering support systems; therefore many studies have concentrated on defining and exploring the meaning and the dimensions of the term *integration* as applied to environments. Wasserman [230], for example, identified five different kinds of integration:

- **Platform**: concerned with interoperability among tools, achieved through the use of a common set of system services;

- **Presentation**: stress on members of a toolkit giving the same "look and feel", via common GUI concepts and design;

- **Data**: sharing data between different tools and handling the data relationships among objects produced by them;

- **Control**: monitoring the tools' operation, and using such information to guide the development process; and

- **Process**: realizing a well-defined software development process, by defining and tracking its steps.

According to this categorization, the work presented in this paper would be categorized mainly as control integration, even though guided by process.

In the attempt to fulfill the various requirements of control integration, and to overcome its inherent difficulties, the software engineering community has developed a wide spectrum of different approaches. Systems and methods are quite numerous, even when one decides — as we do in the rest of this section — to neglect what is probably the largest category: symbiotic *collections* of tools that (as, for instance, in the case of UNIX [133]) are sometimes claimed as environments themselves, although they typically realize only platform integration.

Many methods embrace the White Box paradigm, with great variation among them with respect to the amount of tool code that must be generated or modified to achieve integration. An extreme approach in this sense is the realization of a set of custom tools, all managed by a common framework; typical and well-known examples are language-based environments generated by Gandalf [91] or the Synthesizer Generator [195], where usually tiny tool fragments are organized for execution in an *incremental* fashion as small portions of the program are edited, or interpretive systems such as Smalltalk [87], in which all the tools are combined together at run-time in the memory space of the language interpreter.

For many other environments, the common framework realizing a form of White Box integration of their toolset — focused on the data dimension — is represented by the database where the results of all the development activities, in their intermediate and final stages, are stored and shared. The tools are on the one hand forced to be closely related, since they must be able to use the same data formats, and on the other hand benefit in terms of performance, because they can reuse data produced by other utilities during previous operation. Some example databases intended for use by environments are GRAS [135], based on an extension of the classic Entity-Relationship data model, and Damokles [55], which employs schemas in the form of attributed graphs. Adele 2 [20] enhances this methodology via a system of triggers connected to the state of the database, so that data modification by one tool is recognized and may cause the invocation of others.

The idea of assigning the role of main integration principle to a common object-oriented data repository has been employed quite widely, including by several of the projects aimed to define *standards* for building generic tools with a high degree of portability and interoperability, and therefore widely reusable — although only under the standard's specifications. PCTE [74] is probably the best known of such standards. The goal of PCTE is to create a set of services and facilities, called a public tool interface, complete enough to support tool implementors in very different situations and domains; many environment prototypes and projects (e.g., [222, 36, 82]) already exploit this facility. Another proposed standard that exploits an object-oriented repository for its integration mechanism is the Ada-specific CAIS-A [164].

A different approach to the White Box paradigm, intended to be more cost-effective than building custom toolsets around a given framework, is represented by the class of systems based on *event notification* — whose stress is on control integration rather than data integration. Field [193] is viewed by many as the archetype of this class of system: its basic principle is the addition of interface modules that send and receive specialized messages to the code of generic tools (in some cases this can be achieved by Grey Box extensions or Black Box wrappers). The messages produced by a tool are sent to a centralized component, known as the Broadcast Message Server (BMS), to inform it about the actions performed during the work session. The BMS elaborates them and produces further information that is sent on to other tools, who have registered for that pattern of message without necessarily any specific knowledge regarding the tool that produced it, in order to coordinate their operation.

YEAST [198] is another system using a form of event notification: it also has a client/server structure, in which the server accepts from the clients event pattern definitions associated with action specifications. It is also able to recognize the occurrences of events in the general computer system, such as time passage, timestamp modifications etc., or can be notified of such occurrences, either interactively by users or automatically by tools. In response to an event recognition, YEAST takes the actions that have been previously associated with that event.

Polylith [188] combines an event-driven approach with another technique in the spectrum of White Box integration: tool fragmentation. While entire external tools can be incorporated in Polylith, by relinking with the provided libraries that support the interface to the system's kernel, more often tools are identified with simpler *services* — or modules or subroutines — whose structure is declared in a service database, and whose free combination and communication is used to obtain the performance of various complex, full-fledged applications and to carry out all the tasks supported by the environment. Further, modules are configured in a distributed fashion, and may even be packaged up and moved among hosts during execution [189]. Many commercial *message bus* products, such as Sun Tooltalk, DEC FUSE and HP SoftBench, combine ideas introduced in Field and Polylith.

Tool fragmentation (usually in larger pieces than for the language-based editors above) is the basic integration principle of several systems, including RPDE [93] [173], Odin [43] and IDL [211] [212]. RPDE maintains tables that represent its tool fragments as the cross-product of objects (i.e., structural components that can be manipulated by applications) and roles and methods (i.e., procedural components used to act upon objects). Odin has a very similar concept of objects and of the tool interactions that manipulate them; it also provides a language to specify tasks and composite tools, whose operators are represented by tool fragments and where objects play the role of their operands. Similarly, IDL proposes a notation to define the structural and functional features of its tools, each of which can be seen as a "building block" with a front-end for input, a composite structure defining its algorithm, and a back-end for its output. IDL declarative statements also describe how to connect several of these components into composite tools. The same kind of notation is now used as part of the CORBA distributed computing standard to describe data transmitted among clients and servers [214].

Since White Box, in all of its flavors, is the kind of integration most frequently implemented by environment builders, less work has been done on Grey Box methods. This paradigm does not require any code modification to the tools, which instead must provide an extension language or API, so that functions can be written to interact with the environment. Unfortunately, relatively few applications (aside from database management systems) are equipped with features that allow to build arbitrary functional interfaces to an environment framework. An attempt to address this limitation is presented by Notkin and Griswold [169], who proposed a mechanism to dynamically and incrementally extend the functionality of generic software systems, without modifying the underlying source code.

Mediators have been proposed as a general architectural facility for integration of perhaps legacy applications whose interfaces do not nicely fit together and cannot readily be modified to match [234]. The mediators comprise special "glue" that make whatever transformations are necessary among relatively independent subsystems to make them work together, and often involve callbacks from the glue code to the application or vice versa — which assumes an API on the part of at least one of the several coupled components. This approach has been applied to large environment components such as object-oriented database management systems [233], transaction managers [105], and process engines [224], as well as tools.

We maintain that Black Box integration, via tool wrapping/enveloping (a form of mediation without the explicit API and callbacks), is probably the most flexible and general methodology since its conceptual aim is the encapsulation in the environment of external tools with no changes to their code, nor need for other kinds of functional capabilities.

ISTAR [57] appears to be the initiator of studies along these lines. While it provided its own development and integration toolkit to help construct new dedicated programs according to the needs of a particular environment, ISTAR also allowed use of third-party applications, simply by encapsulating their invocation into the code of ad hoc envelopes that provide the correct interaction with ISTAR's database and user interface.

As we already pointed out in Section 6.2, Oz employs shell-script envelopes to invoke the activities of process tasks and abstractly represents external application programs as object classes in a toolbase. Another example is offered by ProcessWEAVER [67], a commercial system embracing Black Box integration and combining together a message bus and a process engine. ProcessWEAVER models tools as objects of class TOOL, and envelopes have the form of interpreted procedures with a syntax similar to UNIX shell scripts. Most process-centered environments, among those that do not rely on White Box methods, provide a system-specific enveloping language and/or exploit standard scripting languages such as Tcl [175] or Python [231].

Many systems provide some means for off-loading the execution of tools away from where they would "normally" run. The simplest is remote job control, such as UNIX `rsh`, which invokes a program or script on a specified host. It can be used to take advantage of tools that do not operate on the user's machine. Some environments, such as Spice [49] and DSEE [145], automatically distribute tool executions to other hosts on a local area network. Their main goal is to achieve load balancing, e.g., for a large system build. These approaches seem limited to batch tools, such as compilers, with no user interaction. Batch tools inherently do not admit sharing of a single execution instance, except in the degenerate sense that multiple users may happen to want to compile the same version of a file and once is enough, but are easily amenable to Black Box integration methods.

WebMake [8] may be the ultimate combination of remote job control and load balancing, whereby tool invocations can be automatically sent over the Internet to other sites on the World Wide Web that participate in the WebMake protocol by installing a particular program (a "CGI-bin") in their website. The data might reside at a remote site, or the tool might need to execute on a particular machine architecture. Server load is considered, with the possibility of offloading to another host at the same site or back to the originating site, with all necessary data transfers handled transparently. Interactive tools can be invoked, but by delegating control to a resident user at the relevant Internet site rather than sending the GUI back to the originating user. We have recently constructed a Web-based Oz client [56], which is intended to eventually support the same kind of facility.

Various systems support some form of tool instance sharing. XTV [1] is a utility related to *xmove*, but operating at a finer granularity and considerably more sophisticated. It displays the graphical user interface of an X Windows tool to multiple users *simultaneously*, as opposed to one at a time, but still only one user has control of the mouse and keyboard at any given moment. Tools may be integrated (with XTV, not a PCE) in Black Box fashion with no modification or extensions. If we had employed XTV instead of *xmove*,[6] then most of our `MULTI_QUEUE` tools could nominally become `MULTI_NO_QUEUE` as far as MTP was concerned, but still lacking facilities for truly concurrent work. Suite [52] is a toolkit for constructing shared GUIs for computer supported collaborative work tools, where generally the tools must be modified or written from scratch (i.e., White Box). It has been applied to a number of software engineering tools in Flecse [53]. Suite also utilizes floor-passing, as in our `MULTI_QUEUE`, but with the advantage — like XTV — that all users can see the tool's GUI simultaneously.

6.7 Contributions and Future Work

We have fully implemented all the facilities discussed in this paper, except as noted in the text, and support the tools we chose as test cases for MTP's four work models. The completed experiments — all of which run quite satisfactorily — have demonstrated the feasibility of employing wrappers for persistent tools within a process-centered environment framework. We expect that an analogous approach would work for integrating legacy applications into a variety of software development environment frameworks and other kinds of integration architectures.

Further, we have introduced several useful concepts to the domain of Black Box tool integration, including a categorization of tools into families with diverse multi-user and multi-tasking capabilities, the notions of multiple complementary enveloping protocols and of loose wrapping, the idea of interfacing with already-executing persistent instances of programs external to the environment, and the ability to extend the functionality of intrinsically single-user tools to partial sharing of their data and computational resources. The support for directing tool execution to a proxy client, when the `host` or `architecture` field is non-empty, also extends to Oz's original SEL protocol, since the

[6]We chose *xmove* over XTV primarily because the former was developed by another group at Columbia.

pragmatic problems of host licenses and platform dependencies apply even to the relatively mundane tools (compilers and the like) supported by previous approaches to Black Box enveloping.

The `MULTI_NO_QUEUE` model presented here is best suited to *asynchronous* groupware applications, where users enter and leave the tool as they please. There is as yet no facility in Oz to define, as part of the process, the circumstances under which tool sessions should be automatically opened/joined and exited/closed; adding such a feature would still allow for asynchronous groupware but more closely couple sessions with the workflow in a manner similar to how individual activities within those sessions are supported. We have already developed preliminary process support for *synchronous* groupware, in which multiple users perform an activity together at the same time [24]. For example, the `multi-flag` field, originally introduced for MTP, is now used within SEL to identify tools that support this kind of collaboration, so that the system can simultaneously submit the activity and its arguments to the clients corresponding to multiple designated users [21]. We have also recently added support for either a human user or the process to *delegate* control over pending tasks to alternative users [224], as opposed to machines, along with corresponding user interface support (agendas treated as menus to select which of the enabled tasks to do next).

One interesting future direction would be to split off all tool management (for both MTP and SEL) from the Oz server into a separate component, independent from the process engine, that would execute as another operating system process distinct from the server, user clients and proxy clients. This would lower the load on the server, simplify later replacement of the component within the Oz system (if desired), and ease the incorporation of both MTP and SEL facilities into other environment frameworks.

Chapter 7

Federating Process-Centered Environments

Abstract

We describe two models for federating process-centered environments (PCEs): *homogeneous* federation among distinct instances of the same environment framework enacting the same or different process models, and *heterogeneous* federation among diverse process enactment systems. We identify the requirements and consider possible architectures for each model, although we concentrate primarily on the homogeneous case. The bulk of the chapter presents our choice of architecture, and corresponding infrastructure, for homogeneous federation among MARVEL environment instances as realized in the OZ system. We briefly consider how a single MARVEL environment, or an OZ federation of MARVEL environments, might be integrated into a heterogeneous federation based on ProcessWall's facilities for interoperating PCEs.

7.1 Introduction

Large-scale software engineering projects are not always confined to a single organization (e.g., group, department or lab), or even to a single institution (e.g., in a subcontracting or consortium relationship). A project may span *multiple* teams located at geographically dispersed sites connected by a wide area network (WAN) such as an organizational intranet or the Internet. Distinct teams may each have their own software development practices, favored tools, use different programming languages, etc. Yet the teams may still need to collaborate frequently in real-time, i.e., operate concurrently rather than sequentially, share part or all of their code and document base, perform tasks on behalf of each other and/or jointly, and so on.

Note we generally use the term "site" to mean an administratively cohesive domain, in which most (but not necessarily all) machines share a single network file system name space, e.g., cs.columbia.edu, as opposed to either a single host such as westend.psl.cs.columbia.edu, a lab subnet within an administrative domain such as psl.cs.columbia.edu, or a campus backbone such as columbia.edu. However, as we shall see, we sometimes use the term "site" in an alternative sense where a single local area network (LAN) or even a single machine may be home to multiple sites — when multiple teams happen to do their work on that same LAN or machine, respectively. That is, a site is whereever a team does its work.

Consider, for example, several teams each responsible for a separate set of "features", all intended to be included in an upcoming Microsoft product release [48]. Imagine some of these teams have been subcontracted from various independent software houses located outside Microsoft's main development lab, perhaps even outside the United States. Although Microsoft documents recommend vendor processes, it seems unlikely that these teams would follow identical software engineering practices, use exactly the same tools, etc. They may not be willing to publicize (even among themselves) their proprietary software development "trade secrets".

There are various approaches to software development environment (SDE) support for multi-site projects. For the purposes of this chapter, we organize these approaches along two orthogonal axes: tightness of coupling and degree of heterogeneity. At one end of the coupling spectrum, each team chooses its own SDE (which may happen to be copies of the same environment in some of the sites) and there may be more or less concern with whether the different team's SDEs are compatible with each other.

A little further along the coupling spectrum, the teams may choose the same (homogeneous) SDE, to minimize data conversion and supply a common vocabulary, or they may use heterogeneous SDEs but agree on a shared data interchange format. In either of these cases, sharing and collaboration between teams is done *outside* the environment — unless some special "glue" is added on top to bind them together into a federation (i.e., a common data format alone is not sufficient for them to work together at run-time), as explained below.

Another important intermediate range is covered when the teams share the same instance of what we call a *multi-site* SDE, which distinguishes among teams (who may reside at the same or different sites) in some way, but provides facilities for sharing and collaboration between teams *inside* the environment. That is, the glue (or perhaps "cement" in this case) is part of the environment framework itself. The degree of independence afforded each team determines the point within the subrange. The heterogeneous version of this intermediate range consists of *multi-SDEs*, that is, interacting but distinct SDEs with the glue consisting of a shared standard event notification scheme [17] or other control facilities in addition to a common data interchange format.

Finally, the far extreme is a geographically distributed SDE that does not distinguish among teams — all the users are treated as members of one very large team sharing everything. We choose the terms "multi-site" and "geographically distributed" here because many SDEs are said to be "distributed", meaning they have multiple internal components that may execute on different hosts on a LAN or WAN.[1]

[1]Some authors have used the terms "multi-site" and "geographically distributed" interchangeably, but here they refer to different concepts.

The geographically distributed SDE end of the spectrum is analogous to distributed database systems, where there is transparent access to distributed data, while the independent choice of SDE end is comparable to a collection of independent databases. The database community has also delineated an intermediate range, often termed "federated databases" [205, 190]. Federated databases generally permit a high degree of autonomy with respect to one or both of two criteria, schema and system: local components of a single database system with intrinsic federation glue may devise and administer their own schema independently (known as a *homogeneous federation*), and/or the local components may correspond to different database systems from among those supported by extrinsic federation glue (*heterogeneous federation*) — in which case even conceptually equivalent schemas may appear in different forms due to system-specific data definition languages.

We are concerned in this chapter with the subclass of SDEs known as *process-centered environments* (PCEs) [200, 34]. In general, a PCE is a generic environment framework, or kernel, intended to be parameterized by a *process model* that defines the software development process for a specific instance of the environment. The PCE's *process engine* interprets, executes or "enacts" the defined process, to assist the users in carrying out the process by guiding them from one step to another, enforcing the constraints and implications of process steps as well as any sequencing or synchronization requirements, and/or automating portions of the process. A federated PCE might coordinate users from multiple teams working on collaborative tasks, inform one team when it should perform some task on behalf of another, notify one team on completion of some task it has been waiting for another to perform, and transfer process state and product artifacts (design documents, source code, executables, test cases, etc.) among local components of the federation as needed for this work.

It is important to note that in both multi-site PCEs and multi-PCEs, we treat process as the *integrating principle* of federation. That is, the federation is intended to fulfill the semantics expressed explicitly in the (global) process, and this has a crucial impact on the design of the federated architecture. We do not address non-process-centered SDE federations further in this report.

A multi-site PCE is analogous to a homogeneous database federation. In particular, the PCE process model fills the role of the database schema with respect to homogeneous federation: the local components of the multi-site PCE are identical, except that they are tailored by and thus enact different process models (or possibly reflect different instantiations of the same process model). A multi-PCE is analogous to a heterogeneous database federation, and similarly requires that each separate PCE is independently (from the others) capable of interfacing to federation glue that makes it possible for them to work together. Generally the process modeling formalism as well as the process model are different at each site, although the process model could be conceptually the same while expressed differently. In either case, again the process model fills the role of the database schema, although we note that generally each PCE also supports some schema for a data repository containing its software development artifacts and process state.

Figure 7.1 illustrates the space of approaches, highlighting the two "federated" grey areas, which serve as the context of this chapter. That is, we are concerned with federated PCEs that exhibit at least some degree of coupling but also at least some degree of independence, i.e., not transparent distribution; and we do not consider completely homogeneous approaches, where even the processes must be identical, or completely heterogeneous approaches, where it is impossible to introduce any sort of run-time integration, and the only possible integration is at definition-time, through a process definition exchange format, e.g., as promoted by WfMC [157].

A federated PCE for cross-organization projects should permit each team to specify its own local process model, along with the desired collaboration with other teams through shared subprocesses, tool sets, data subschemas, data instances, etc. Thus, a noticeable difference from database federation is that the focus here is on interoperability among heterogeneous processes, i.e., the application semantics, as opposed to (only) heterogeneous data schemas, i.e., the data on which applications operate.

One approach to homogeneous PCE federation, where every team runs a component of the same multi-site PCE but enacts a different process, is taken by our Oz PCE [27]. Oz was devised to scale up our earlier MARVEL PCE [123] to multi-process, multi-team, geographically dispersed software engineering projects. Oz introduces an *International Alliance* metaphor whereby each team

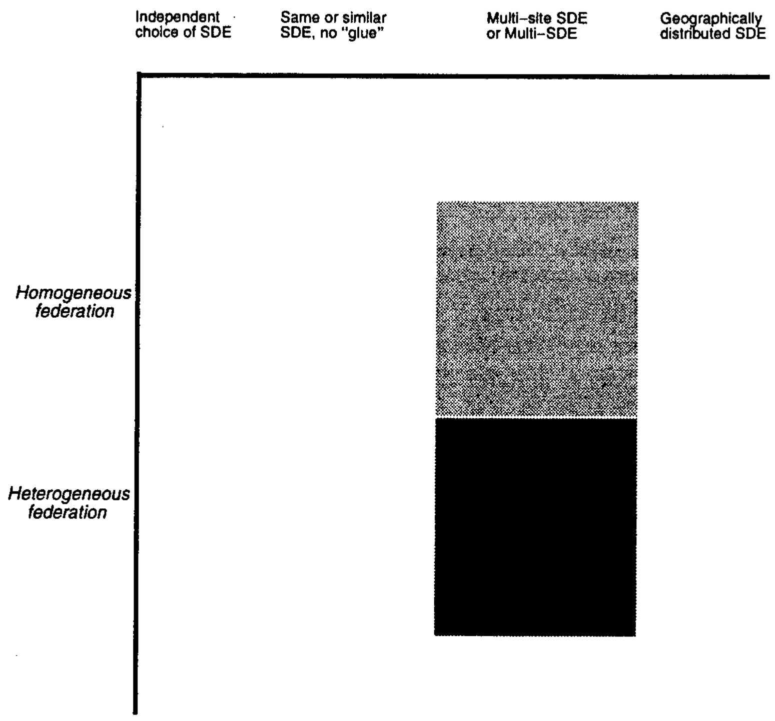

Independent choice of SDE
Same or similar SDE, no "glue"
Multi-site SDE or Multi-SDE
Geographically distributed SDE
Homogeneous federation
Heterogeneous federation

Figure 7.1: Multiple Team SDE Spectrum

autonomously devises its own local process (supported by a local Oz component that is essentially an extended instance of MARVEL), analogous to how each country has its own local customs and laws. A team may agree to extend its process to a small degree (and thus temporarily lose some autonomy) in order to participate in a *Treaty* with one or more other teams. The enactment of a *multi-site task*, defined as any task that involves interaction among the several sites of a multi-site PCE, is called a *Summit*. Oz extends MARVEL with Treaties, Summits, and an underlying inter-site communication and configuration infrastructure where each site corresponds roughly to an instantiated MARVEL environment.

Only tasks specified in a Treaty may access data from other sites, and even then only in accordance with the privileges granted by the Treaty. For example, a site may agree to perform certain tasks requested by another site on its own local data; or a site may agree to allow another site to perform certain tasks on its local data; or a site may agree to perform certain tasks on data from several sites. However, each site (or team) is responsible for any prerequisites or consequences of such tasks with respect to its own data, following its own process, just as in preparations for and follow-ups of meetings among country leaders (the basis for our metaphor). Treaties may be dynamically defined while the process unfolds, i.e., while computation is in progress, permitting a degree of flexibility not found in most distributed systems.

One approach to heterogeneous PCE federation, where two or more distinct process systems are bound together into a multi-PCE, is taken by Heimbigner's ProcessWall [98]. Note that ProcessWall is the external glue supporting such binding, not itself a PCE. ProcessWall could of course be used to integrate multiple instances of the same PCE, say MARVEL, with different process models as in Oz, but in this chapter we address only the more challenging case of using it to federate multiple distinct PCE systems.

Heimbigner refers to ProcessWall as a *process state server* because it enables interaction between the PCEs through a centralized representation of global process state that the teams agree to share. However, we believe it is more useful to treat the mechanism Heimbigner describes as a *process task server*: it may maintain the history of tasks that have already been completed, in aggregate representing the current process state, but more significantly from the viewpoint of federation the server posts those tasks that have been instantiated but not yet scheduled for enactment by one of the participating PCEs.

In particular, each participating PCE manages, schedules and enacts its own task descriptions, usually forwarding each description to ProcessWall only after that task has been completed, e.g., to allow users to exploit ProcessWall's process state inspection facilities (part of the glue). Thus the process remains primarily decentralized, since the actual process operation is performed by the separate PCEs without any interactions between them or with ProcessWall while the work is in progress. However, in some cases a PCE may send an instantiated (e.g., with data parameters) but unenacted task to ProcessWall intending it to be executed by some other PCE in the federation, because the sending PCE does not have the data, tool(s) or user(s) appropriate to conduct the work.

An intelligent scheduler might then be attached to ProcessWall to direct such posted tasks to particular sites, as described in [174], or alternatively ProcessWall might be treated as a "blackboard" (using artificial intelligence terminology [95]) from which the schedulers of the individual PCEs participating in the federation select those tasks they are suited to perform. Any sharing of software product artifacts, as opposed to process state, is implicit in the data information included with posted tasks. As in OZ, each site might autonomously devise its own process model.

Mentor [232] is similar to ProcessWall but divides the process state/task server into two components: a *worklist manager* acting as a pure task server and a *history manager* corresponding to a pure state server; data sharing is factored out as in ProcessWall. Note Mentor is a workflow management system intended for business applications, not a PCE oriented towards software engineering; whether there is any fundamental difference between workflow and process is a matter of some debate [204], but we blur the distinction in this report. In any case, heterogeneous federation based on Mentor would probably be quite similar to the ProcessWall model.

Process interchange formats [157, 146] support translation of a logically single process model into the different representations of distinct process systems, but do not provide any means for collaboration and interoperability during the process enactment by those systems. Thus there is no true federation in the sense addressed by this chapter. However, some kind of translation facilities are needed as part of any heterogeneous federation: Mentor transforms the heterogeneous process modeling formalisms into StateMate charts [112], but in the case of ProcessWall only process state is translated (or the participating PCEs might be implemented to use a common task format).

We mentioned above that process enactment by a federated PCE might involve movement of product artifacts among teams that could potentially be distributed across a WAN. Alternatively, all the sites might share a common centralized data repository, presumably located at one of the sites, or even a transparently distributed data repository. Globally shared data seems most appropriate for projects organized far in advance and involving only a single institution, perhaps with multiple campuses. In contrast, when different institutions work together, particularly when the federations are dynamically created and dissolved, most likely the institutions would prefer to maintain locally at least those product artifacts produced by their local process.

This chapter discusses the architectural aspects of PCE federation and associated infrastructures, and then justifies our architectural choices for the fully implemented (and used in our day-to-day work since April 1995) OZ system in detail. We also explore a hypothetical OZ/ProcessWall interface. Our investigation of architecture is strongly influenced by the fact that the main purpose of PCE federation is to enact multi-site and global processes. For example, global processes devised using a top-down methodology, say intended for multiple campuses of a single institution, may require somewhat different architectural support than global processes constructed in a bottom-up manner, e.g., for temporary multi-institution collaborations. However, methodologies for developing global processes are outside the scope of this report.

First we present architectural requirements and the alternative architectural models we considered for both homogeneous and heterogeneous PCE federations, the latter in contrast to the former (i.e., many of the requirements are shared and the architectures are analogous). We then elaborate the specific design decisions and tradeoffs that were made in developing the OZ architecture and infrastructure that extended our earlier MARVEL PCE to a homogeneous PCE federation. We do not go into detail regarding ProcessWall, Mentor, or any other such heterogeneous federation glue, since that is properly left to their developers. Instead we briefly discuss how MARVEL, or OZ, might be integrated into a heterogeneous federation based on ProcessWall's process state/task server model, to some extent synergizing the two federation mechanisms, i.e., allowing OZ to operate as a multi-site PCEs within a multi-PCE. In both sections, the range of architectures is explored specifically in the context of our choices for OZ. We conclude with the contributions of this work and outline some directions for future research.

7.2 Requirements and Alternative Architectures

In both the homogeneous and heterogeneous federated models, each local site runs a component of a multi-site PCE or multi-PCE. We refer to such a site component as a *sub-environment*, or just SubEnv, even though it may operate in stand-alone fashion as a full PCE. We refer to the "glue" that holds the SubEnvs together as the federation's *foundation*, or just Foundation. Database federation involves a similar foundational component or layer, e.g., to control global transactions, although many classes of distributed system do not include any foundational layer beyond a basic networking communication protocol. This section of the chapter is concerned with the functionality (Section 7.2.2 for the homogeneous case and Section 7.2.5 for the heterogeneous case) and architectural design (Sections 7.2.3 and 7.2.6, respectively) of the Foundation. Recall that we are mainly concerned with multiple sites on a WAN, generally with independent administrative domains — although of course nothing prevents multiple sites from running on the same LAN, that is, a multi-site PCE or multi-PCE might operate entirely within a single organization or group and each "team" could conceivably consist of only one user (as in the OZ EmeraldCity environment we use to support our own software development [125]). We take as given the requirement that each site must be able to support an autonomously devised process model.

7.2.1 Local Environment Internal Architecture

Although the focus of this chapter is on *federation* architecture, it is useful to begin the discussion with an overview of SubEnv *internal* architectures, since they have a substantial impact on the design of a homogeneous federation; internal architecture is less germane in the case of heterogeneous federation since, in general, each participating SubEnv may employ a different internal architecture. As we focus in this chapter on process-centered SDEs and the impact of process on architecture, we characterize local PCE architectures based on the degree of centralization in process enactment, comprised of two aspects: process control and tool execution. The former refers to the function of deciding which task to enact, when, according to process constraints/context, whereas the latter corresponds to where and how the task gets executed, often but not necessarily via one or more specific tools. This separation is important in PCEs because it reflects the typical separation between the process itself and the tasks spawned by it, which may invoke external tools, take arbitrarily long to complete, involve one or more human (possible simultaneous) users, and so on.

Although our goal was to scale up our pre-existing MARVEL PCE to support multiple teams each sharing a potentially different process, where the teams might be connected by either a LAN or a WAN, we identified four classes of internal PCE architecture — only one of which applies to the final MARVEL version 3.1.1 we were concerned with. Note these are not the same classes suggested by Peuschel and Wolf [180] and we follow a different classification scheme: Peuschel and Wolf were concerned with the relationship between the process engine and the data repository, whereas we consider process control vs. tool (or task) execution.

1. **Centralized process control and Centralized tool execution:**
   This is the simplest case, where both control and execution are carried out by the same component. An all-in-one single-user PCE such as MARVEL 2.x [119] and some compiled process programs, e.g., written in APPL/A [219], would fit into this class. Even a client/server system might fall into this category if the client supported only the user interface and all process

enactment was performed in the server. Given the multi-user multi-task nature of practical software engineering processes, this architecture is inherently unscalable, even for a single team.

2. **Centralized process control and Decentralized tool execution:**
A process server maintains the state of the process, controls its enactment, and synchronizes access to shared resources, but the tools themselves execute at process clients. MARVEL 3.x [31], ProcessWEAVER [67], and Mentor fit this mold, albeit in different ways. MARVEL 3.x relies on fixed user clients to fork tools, whereas ProcessWEAVER spawns user "work contexts" as needed by the process. OZ local sites are somewhere in between, with one server per site (i.e., per team), generally employing user clients as in MARVEL 3.x but also supporting "proxy clients" that run tools on behalf of one or more users under various circumstances, as explained in [227, 209]. Mentor is similar to OZ in that user clients can connect to multiple process servers in the federation.

3. **Decentralized process control and Centralized tool execution:**
Control is distributed among multiple process servers, where the tool execution function is supported by a single component. This model supports separate process engines for each user — as in Endeavors [35] or Merlin [201] — while sharing special computational or database resources used in tool invocation. One can easily imagine multiple workflows accessing the same tool management resource, particularly if only the tool broker is centralized, directing actual tool invocation to distributed hosts as in WebMake [8].

4. **Decentralized process control and Decentralized tool execution:**
Here the process itself is distributed across multiple nodes, where each node is responsible for the execution of its subprocess as well as corresponding tasks. Control flow and synchronization between the process segments is specified locally inside the nodes. Several transactional workflow systems, such as Exotica [5] and Meteor [116], operate in a fully distributed manner — by expressing the workflow implicitly in a network of task managers (which invoke the actual tools) that interact only with their predecessors and successors in the workflow routing.

Several issues influence the choice of single-site PCE architecture. A major factor is the level of data integration employed by the PCE for product artifacts. PCEs with extensive data integration facilities (e.g., SPADE [10], EPOS [45]) might choose a centralized control architecture to minimize communication between the data and process managers when disseminating tasks — unless the data management is itself distributed, and/or the data itself is physically distributed, in which case a distributed control architecture may be employed. PCEs with no data integration facilities might be fully distributed in an easier manner. Note, however, that full distribution of process enactment is not incompatible with sharing a centralized data repository; see [180].

Another characteristic that impacts the choice of local PCE architecture is whether the process modeling paradigm employed by the PCE is reactive or proactive (termed proscription vs. prescription by Heimbigner in [96]). Reactive enactment may be realized better in a centralized-control architecture, as requests for enactment are directed to a single server that dispatches the service to a client (perhaps the requester itself), whereas proactive enactment may be distributed by assigning a priori each task to a component, with the ordering and execution constraints inside each component — or implicit in their interconnection topology.

7.2.2 Requirements for Homogeneous Federation

We have identified the following additional requirements for the homogeneous model:

- Communication infrastructure.
The most fundamental functional requirement for multi-site process enactment is that the Foundation include an infrastructure whereby the SubEnvs communicate with each other regarding multi-site tasks. This might be constructed directly on top of TCP/IP sockets, or

employ some higher level mechanism such as RPC or CORBA [172]. In any case, we are mostly concerned with the PCE-cognizant interconnectivity layer, i.e., the Foundation, not the underlying mechanism.

- Global process definition and acceptance.
  On top of the basic interconnectivity support, the Foundation must supply means for local processes to interoperate, i.e., to model and enact tasks that in some way span multiple processes/sites and contribute to the global process. In particular, the Foundation must have facilities to (re)negotiate and (re)define (possibly dynamically) the specifics of process-interoperability for the relevant processes, e.g., via Oz-like Treaties.

  Although a degenerate global process may involve only primitive operations (e.g., copy a data item), we in general assume some notation to define multi-site tasks whose enactment is controlled to some degree by the Foundation. In other words, we assume that multi-site tasks are themselves modeled in either top-down or bottom-up fashion as parts of a global process, with conceptually its own state and purpose. However, multi-site process modeling and enactment is the subject of another chapter; here we are concerned with structure and organization of components, i.e., architecture, that supports multi-site processes.

- Local autonomy and independent operation.
  As far as purely local work is concerned, i.e., work involving the local process operating only on local data, a SubEnv should operate autonomously and independently, and provide the same capabilities as would a single-site PCE. It should not in any way rely on communication with other SubEnvs, or with the Foundation, in order to perform its standard functions with regards to defining and executing the local process. The underlying assumption is that most of the work done by a site is local to that site, and therefore the multi-site PCE should still be optimized towards local work.

- Restrict global dependencies to affected sites.
  A related issue is that the SubEnv should minimize the dependencies on uninvolved SubEnvs when executing part of a multi-site task. These two requirements are somewhat similar to control and execution autonomy, respectively, in multi-database transaction management [138]. The local site autonomy prized in the Oz approach to bottom-up process modeling has also been argued as necessary for top-down modeling: "A participant on a lower level [of the hierarchy] does not want his/her management to know how a task is performed" [202]. Thus we rationalize site autonomy as a critical requirement.

- Configuration awareness.
  The SubEnvs must somehow be aware a priori (statically), or become aware during the course of process enactment (dynamically), of each other's existence, i.e., the other members of the currently configured federation, if they are intended to directly communicate and, possibly, collaborate. In the case where a Foundation intermediary is the conduit for all communication and interactions among SubEnvs, the SubEnvs must at least be aware of that intermediary and vice versa.

- Dynamic (re)configuration.
  Since the lifetime of enacted processes is often long, months to years, the Foundation must allow for SubEnvs to join or leave a federation while a process is in-progress, that is, support configuration and reconfiguration of participants in the global process. It is of course also necessary for SubEnvs to determine or negotiate what services each can expect from other (perhaps anonymous) SubEnvs in terms of process control, tool execution, and data and other resources, and how to coordinate exploitation of those services, but again that is the subject of another chapter.

- Process state survives failures.
  Since processes in general, and federated processes in particular, are enacted for long durations,

they require facilities for persistent process state. In cases where each local PCE manages its own product-data repository, the Foundation must also provide mechanisms for transferring product artifacts, in addition to process state, among sites. This may involve the same or different inter-PCE communication channels for product vs. process data, but the two cases have to be handled separately because products typically involve significantly larger volumes of data. For example, in a multi-site `build` task one site may collect code modules from the other relevant sites and return to them copies of the resulting executables and/or libraries. Another example is a distributed groupware task such as multi-user editing, in which source code and/or documentation files stored at one site may need to be (simultaneously) transferred to several other sites. In general, bulk data may be temporarily cached, permanently copied, or migrated between sites.

- Semantics-based transaction management.
  Another data-related requirement involves support for sophisticated and flexible concurrency control and failure recovery mechanisms due to the long duration of tasks and task segments, interactive control by users, and human-oriented collaboration among tasks and task segments while they are in progress [139]. The explicitness of the process in PCEs makes it possible to employ semantics-based transaction management [14, 107]. Multi-site tasks may modify data from multiple sites, and thus require some kind of global transactional support, such as two-phase commit that interfaces with local transaction managers. Investigation of this topic is beyond the scope of this report, see [23, 102].

7.2.3 Homogeneous Federation Architectures

We identify five categories of architectures within the homogeneous (light grey) area of the "tightness of coupling" spectrum of Figure 7.1. Note relatively minimal (or no) translation services are needed in any of these categories: all the SubEnvs speak the same languages (including data formats, process modeling notation, and tool wrapper scripts). The Foundation may perform name mappings, since a common ontology is not assumed, but this is not its major function.

For the sake of the figures depicting the multi-site PCE architectures below we show one plausible set of components that may comprise a local SubEnv (process, data, tool, and user interface components), but we do not intend in any way to specify or constrain a SubEnv to follow the given structure, and any of the internal architectures discussed in Section 7.2.1 might be employed. Further, we do not specify any particular internal component for interfacing to the Foundation — interpret the figures as if they *all* (potentially) do, to achieve federation of their various functionalities.

7.2.3.1 Ad hoc

Two or more instances of the single-site PCE are hardwired together in some ad hoc fashion for a particular purpose. There is generally no Foundation, per se. This model obviously does not scale, so is not addressed further.

7.2.3.2 Centralized Glue

The SubEnvs communicate and interact through a single centralized glue component that constitutes the Foundation. As mentioned earlier, the Foundation, i.e., the federation glue, is intrinsically part of the multi-site PCE rather than imposed externally. However, each SubEnv necessarily includes code to interface to the Foundation, perhaps through RPC or TCP/IP socket calls originally part of native SubEnv if the PCE was designed as a multi-site PCE, or inserted later if not. The Foundation may perform brokerage or routing among SubEnvs, and maintain the state of multi-site process segments.

Figure 7.2 illustrates this architecture. Distributed systems with one or more central components do not scale beyond a certain level, since the centralized component becomes a performance bottleneck and single point of failure (i.e., if this one component fails multi-site tasks become impossible). The interface aspect of the centralized glue could be expanded in several different ways with

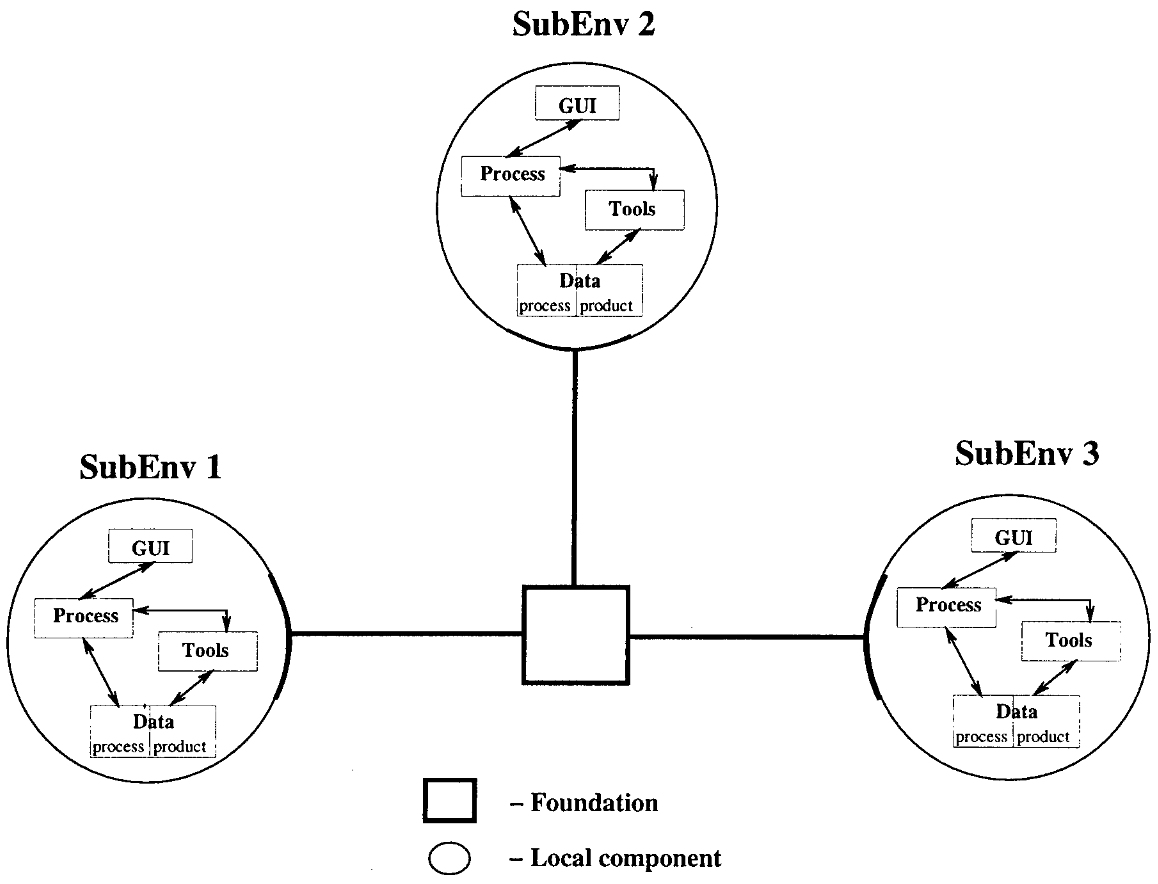


Figure 7.2: Centralized Architecture

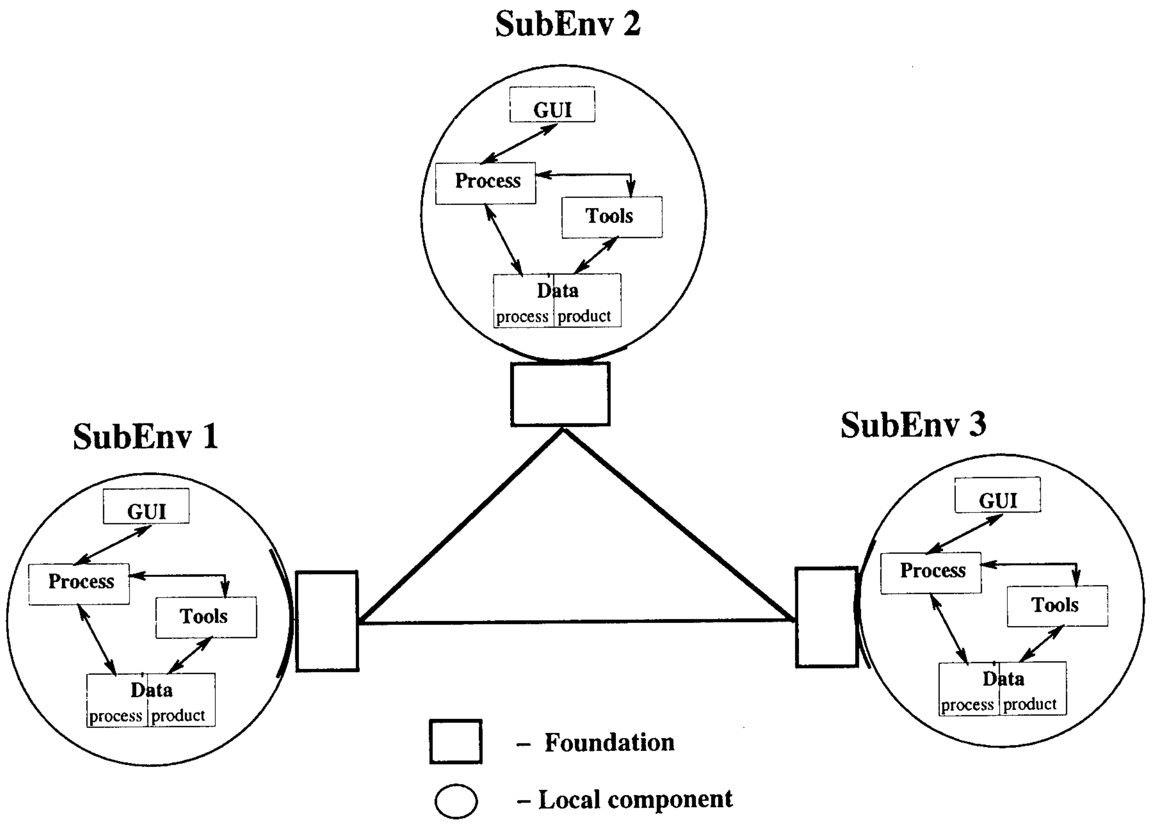


Figure 7.3: Decentralized Glue Architecture

respect to the SubEnvs, analogous to the intermediary, moderated and direct decentralized cases below. We do not discuss these options, since the variations between the cases are overwhelmed by Foundation centralization — although as the interfaces get "larger" the central component tends to get "smaller", as functionality is shifted, effectively achieving a hybrid between centralized and decentralized approaches.

7.2.3.3 Decentralized Glue

The SubEnvs communicate and interact through *intermediaries*, with one intermediary attached to each SubEnv. These intermediaries collectively constitute the Foundation glue, and there is no centralized component. A Foundation intermediary may or may not be realized as a separate operating system process from its local SubEnv. If separate, it would usually reside "close" to the local SubEnv, e.g., on the same LAN, but not necessarily on the same host.

However, this case is distinguished from the peer-to-peer cases below in that the intermediary has no special knowledge of the PCE's process-oriented functions and no access to its process model, nor any special knowledge of its tool execution facilities, data repository, user interface, etc. To the degree that these internal functionalities (whether or not distinguished as components) interoperate within the federation, and thus interact with the Foundation infrastructure, they must interface to the intermediary. The intermediaries are tightly coupled with each other, e.g., maintaining long-term connections which permit them to share the Foundation's global process state and work closely together to realize the Foundation's functionality (e.g., a distributed name service), but are loosely coupled with respect to their SubEnvs. See Figure 7.3. From a process perspective, the interaction between the global process and the local processes is quite limited because the Foundation has no access to the internals of the local processes.

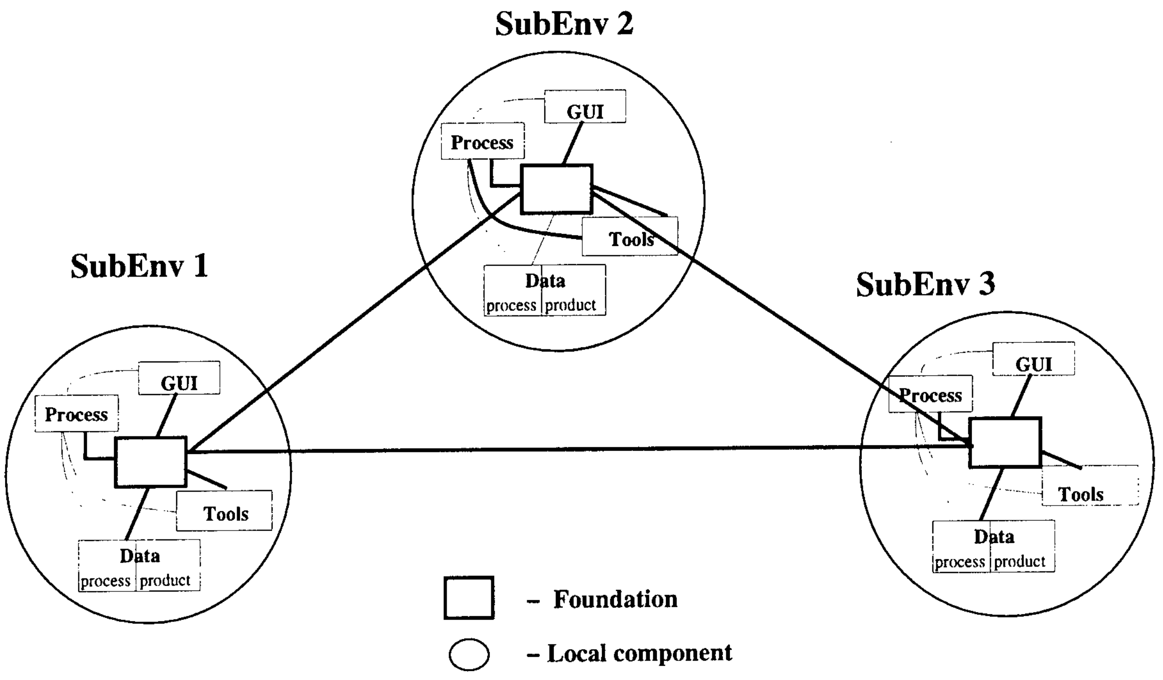


Figure 7.4: Moderated-peer-to-peer Architecture

Note a geographically distributed realization of the decentralized glue architecture is plausible — with the intermediaries acting as gateways to remote SubEnvs on a WAN; the two peer-to-peer architectures below also easily admit a geographically distributed implementation. Although a centralized architecture might also be geographically dispersed, this seems less likely from an administrative point of view — except possibly within a organizational intranet where the same organization owns and controls all the relevant sites including the machine hosting the central component.

7.2.3.4 Moderated Peer-to-Peer

The SubEnvs again communicate/interact through intermediaries, which we call *moderators* here, with one moderator attached to each SubEnv. These moderators collectively constitute the Foundation, and again there is no centralized component. Again there is no implication intended regarding physical realization, the moderator may or may not be realized as a separate operating system process from the rest of its local SubEnv. If separate, again it would necessarily reside "close" to the local SubEnv, most likely on the same host.

Unlike the decentralized glue case, here each moderator is tightly coupled with its local SubEnv and has intimate knowledge of that SubEnv's process-oriented expectations regarding services from other SubEnvs. Similarly, the moderator is cognizant of the local process model and state, tool execution, data repository, user interface, etc., if relevant to federation. Again, to the degree that these internal functionalities (whether or not distinguished as components) interoperate within the federation, and thus rely on the Foundation infrastructure, they must interface to the moderator. In contrast, the moderator is loosely coupled with respect to its peer moderators, e.g., making only short-term stateless connections. See Figure 7.4.

This approach again seems obviously more likely to scale than a centralized architecture, but more-or-less equivalent with respect to scaling as the decentralized glue case. However, in this case the architecture cannot assume any shared capabilities (e.g., name services) provided by the Foundation. In other words, it is a "shared nothing" architecture as far as the Foundation is concerned. (Note this does not preclude sharing among internal components of the SubEnv.) On

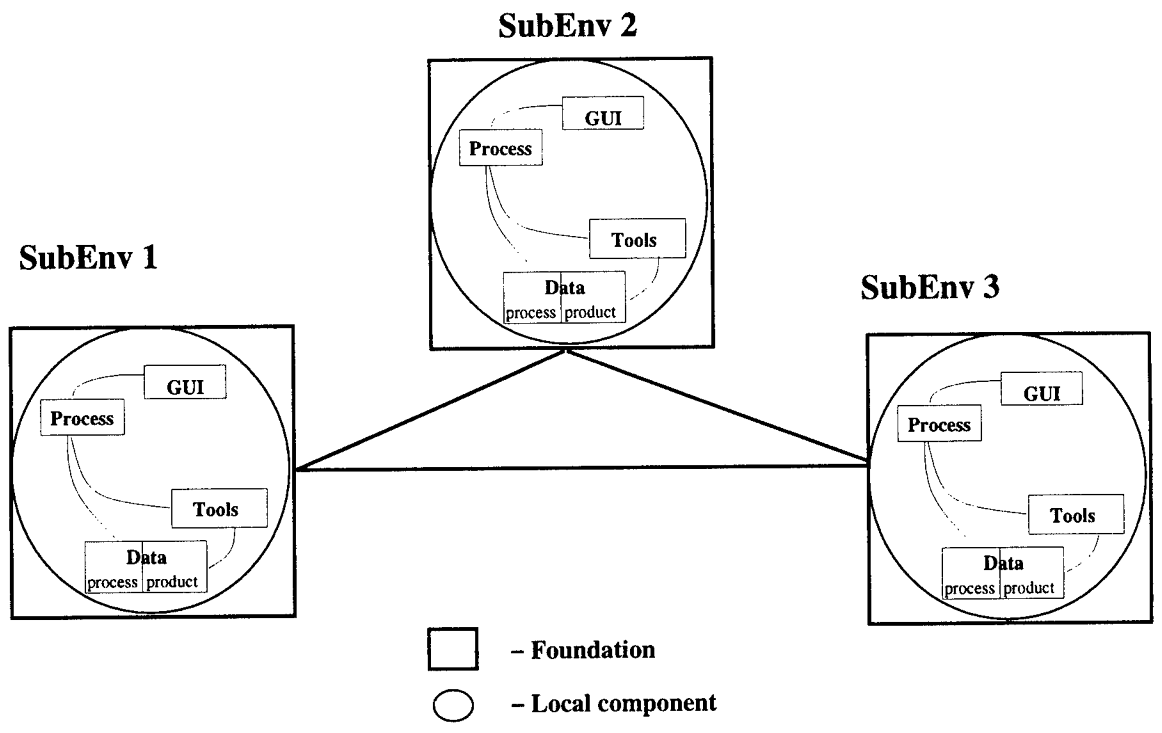


Figure 7.5: Direct Peer-to-peer Architecture

the other hand, the interaction between the global process and the local processes is richer, because the Foundation has direct access to local processes.

That is, the primary distinction between the decentralized glue case and the moderated peer-to-peer case is that in the former the local Foundation components have no knowledge of the local processes and manage a multi-site process imposed on the local SubEnvs divorced from their local processes, whereas in the latter the local Foundation components have intimate knowledge of the local processes, but without any shared global process state or common control. This reflects the tradeoff between stronger coupling within the Foundation and weaker coupling between the local component of the Foundation and its local SubEnv, in the decentralized glue case, vs. weaker coupling within the Foundation and stronger coupling between the local Foundation and its SubEnv, in the moderated peer-to-peer case.

7.2.3.5 Direct Peer-to-Peer

The SubEnvs communicate/interact with each other directly, and the Foundation cannot easily be distinguished from the rest of the multi-site PCE. That is, the local component of the Foundation is *built into* one or more of the SubEnv's internal components, most likely the process engine; there is no specific component introduced *solely* to represent the Foundation infrastructure. See Figure 7.5.

While this approach probably offers improved performance over the others described above, it is more challenging to realize for pre-existing single-site PCEs because it generally involves significant modification throughout the PCE code as opposed to "adding on" interfaces to a new component. Thus scaling is restricted for software engineering rather than distributed computing reasons.

7.2.4 Choice of Homogeneous Architecture

The choice of federation architecture for homogeneous multi-site PCEs depends largely on two concerns:

1. The paradigm chosen for modeling and enacting federated processes.
2. The style, design and implementation of the local SubEnv framework.

Regarding multi-site or global processes, we distinguish between two major paradigms, top-down and bottom-up, although of course hybrids are possible. *Top-down* refers to a process broken down through multiple levels of granularity each corresponding to subsequently smaller organization units, as in the enterprise-level to campus-level to department-level to group-level of the Corporation metaphor [207]; this is analogous to a global transaction in federated databases [186]. *Bottom-up* refers to interoperability among possibly pre-existing local processes, as in Oz's International Alliance metaphor, without a global overseer (unfortunately, the kind of distributed computing scenario where the Byzantine Generals problem arises — although consideration of fault tolerance in the face of malicious behavior is outside the scope of this work). We do not consider here which of the two paradigms is more appropriate for various applications (see [28] for such a discussion), but rather which *architecture* best supports each of the paradigms — particularly the bottom-up paradigm, since one of our major goals was to link pre-existing single-site MARVEL processes.

In order to support top-down global processes, the federation must support maintenance of global process state. This suggests a glue architecture, particularly centralized but also decentralized, where the Foundation manages the state. In contrast, bottom-up federation can naturally be realized on top of a peer-to-peer architectural style, again in one of two possible ways, namely the moderated or direct peer-to-peer architectures. In other words, we make a primary distinction between top-down vs. bottom-up process interoperability, and a secondary distinction between the architectural realization of each style. In general, bottom-up interoperability is more scalable than top-down, as in any other distributed system, but introduces process-related problems in our context such as lack of explicitness of the global process.

The association of top-down processes with glue and of bottom-up processes with peer-to-peer architectures is not exclusive, however. It is potentially feasible, for example, to realize a top-down process using a peer-to-peer architecture, but it is likely to be inefficient and harder to realize because of the needs to distribute the global process state among the loosely coupled intermediaries and to manage shared information over a shared-nothing architecture. It is probably easier to realize a bottom-up process using a glue architecture, provided that administrative barriers (regarding access to private process state at remote sites) can be relaxed or overridden.

Let us now consider the impact of the local SubEnv architecture on the choice of federated architecture. One factor stems from the degree of openness and extensibility of the process control and tool execution components (the data repository is also of concern, but those issues are not terribly different than in other federated database applications, so we concentrate here on PCE-specific matters). In particular, peer-to-peer architectures demand tighter integration at the process control and task/tool execution levels, between the local SubEnv and its Foundation component, than glue architectures. This is possible only if the local SubEnvs provide suitable application programming interfaces (APIs) for extending these functionalities — which would usually suffice for moderated peer-to-peer. Or, alternatively, if the SubEnv source code can be internally modified — which would by definition be required for the direct-peer-to-peer, assuming the PCE was not originally implemented as a multi-site system. (We know of none that were, e.g., multi-site Oz was realized by adapting the single-site MARVEL 3.x PCE.)

Another important factor that impacts mainly peer-to-peer architectures is the degree of centralization of the SubEnv internal architecture. SubEnvs with centralized (local) process control naturally lend themselves to a direct-peer-to-peer federation architecture, where the Foundation infrastructure is built into the local process engine — which becomes the conduit to communicate with other SubEnvs; communication via a centralized tool manager is also conceivable. Fully decentralized (local) process enactment, in contrast, seem better suited to a moderated-peer-to-peer architecture since there is no one component that stands out as the focal point. Instead, a new moderator component is attached to the SubEnv as a whole and communicates with each of the other local components as well as with its peer moderators. However, a direct-peer-to-peer architecture is not inconceivable for decentralized SubEnvs; see [229].

To summarize, the above categories represent different degrees of (de)centralization of the Foundation, ranging from a logically and physically centralized architecture, to several forms of logically and physically decentralized architectures with variations in the coupling between and within SubEnvs. Our key observation is that there is no one architectural style for federated PCEs that is inherently superior to all others. Instead, we argue that the choice of a proper architecture depends on the requirements of the system, and more specifically on the architecture of the local PCE and on the federated process paradigm. This is elaborated for the case of Oz scaling up MARVEL in Section 7.3.

7.2.5 Requirements for Heterogeneous Federation

Recall that in the heterogeneous model, each site (or team) runs a separate PCE that works together with other PCEs in a multi-PCE joined together via the Foundation. Each site may employ a *different* local PCE, selected to best fulfill its own needs or retained for historical reasons. A few may happen to use independent copies of the same system, or local PCEs with similar architectures and interfaces, but we cannot count on that and therefore treat each SubEnv as unique within its federation.

We have identified the following requirements for heterogeneous federation. In general these are *in addition to* homogeneous federation requirements, although in some cases we repeat the seemingly identical requirement followed by new discussion oriented towards the special circumstances of the heterogeneous case.

- Communication infrastructure.
  The most basic function of the Foundation is to communicate with each SubEnv participating in the federation. In general, the SubEnvs cannot communicate directly with each other since (by definition) they were designed as independent PCE systems (although perhaps following a "standard" Foundation interface). Realization of multi-site tasks, or fulfillment of a request from one site for another site to undertake a task on its behalf, requires that some logically homogeneous component is added to each PCE so that it can bind into the federation. This component may involve quite diverse per-PCE physical implementations, e.g., to perform PCE-specific protocol conversions and data format translations. The conceptually common component is relatively limited, though, and in particular does not take over local process modeling and enactment functions — since otherwise we could consider it to effectively convert the federation to the homogeneous case.

- Federation awareness.
  The SubEnvs (usually) must somehow be made aware of any federation(s) in which they participate, possibly more than one at a time, in order to interoperate and contribute to global process enactment. The SubEnvs need no direct knowledge of the other SubEnvs in the federation, per se, but there must be some means whereby the Foundation coordinates the global process, either by *notifying* a given SubEnv that it should or could perform specific tasks or by posting the request to some standard forum that each SubEnv *polls* to choose tasks it is able and willing to perform.

  Note this does not necessarily assume that SubEnvs have some means to inform the Foundation of pending tasks that they are unable or unwilling to do themselves: The Foundation could itself impose all tasks, perhaps through a special process modeling and enactment system intended to act as a "global hand" supporting some form of "superworkflow" [202], analogous to multi-part transactions submitted to heterogeneous multi-databases.

  In principle, it might be plausible for a SubEnv to perform work on behalf of a federation without ever noticing that the heterogeneous federation exists (which would not normally be the case for homogeneous federations). Thus an alternative is that only the Foundation is aware of the various SubEnvs, and picks up their results through some non-intrusive manner, such as understanding file formats of what the PCE considers internal process state information (as done, for example, in [104, 183]). This alternative model operates more in the vein of a broadcast message server, such as Field [194], where the only purpose of the Foundation is to forward notification messages that a particular SubEnv has *already* performed a particular task. This could be augmented with limited process support, as in Forest [79] and Provence [140], to transform notification messages into requests to perform various tasks triggered by process enactment in the Foundation.

- Data translation and transfer.
  When process enactment at one SubEnv involves access to data "owned" by one or more other SubEnvs, the Foundation must provide mechanisms for transferring product artifacts and requisite process state among SubEnvs. As in homogeneous federations, bulk data may be temporarily cached, permanently copied, or migrated between sites. Note that enactment of such tasks may not be frequent in a multi-PCE, e.g., data exchange may be limited to scheduled milestones, whereas collaborative tasks are expected to be more commonplace in a multi-site PCE.

  A homogeneous federation can assume a standard data repository, although perhaps with differing local schemas, whereas heterogeneous federations also incur the problems of incompatible data formats; this is basically a distributed computing issue attacked by OMG [171] and others through CORBA and similar layers, and not addressed further in this report. Further, homogeneous federations assume compatible transaction management (concurrency control and failure recovery), generally supporting a two-phase commit protocol for distributed transactions, which may not be straightforward in the heterogeneous case. This issue has been addressed extensively in the database community, e.g., [40], and is also not discussed further here.

- Federation configuration.
  Finally, there should be some means for configuring a federation. We anticipate this is considerably more difficult and heavyweight for heterogeneous than for homogeneous federations, and in the former case may involve substantial design and implementation to introduce a new PCE (i.e., if it was not previously integrated with the Foundation) rather than just invoking a pre-defined (re)configuration process. Substantial effort may be involved in introducing the conceptually homogeneous infrastructure component mentioned above into a given PCE, and the mechanism for doing so is necessarily ad hoc (i.e., PCE-specific).

7.2.6 Heterogeneous Federation Architectures

The three main categories are outlined below.

7.2.6.1 Ad hoc

A handcrafted federation consisting of a very small number of distinct PCEs, e.g., Bull's ACME [7] integration of ConversationBuilder [130] and MARVEL 3.x. While one might be able to find a special-purpose Foundation component in this or similar examples, we are concerned in this chapter with general federation. There is no distinction between the ad hoc approach and peer-to-peer architectures in the heterogeneous case, because by definition there is no common means for introducing a tightly coupled Foundation moderator, or equivalent code inside the SubEnv's internal architecture, nor any general way for a moderating component to incorporate SubEnv-specific knowledge of process and other concerns.

7.2.6.2 Centralized Glue

The SubEnvs communicate/interact through a centralized component that implements the Foundation — the same architecture as shown in Figure 7.2, except that the SubEnvs may be different (originally) single-site PCEs rather than components of the same multi-site PCE. This approach is exemplified by ProcessWall. Mentor takes a similar tack, except that there may be *multiple* state servers and task servers, not just one (of each).

7.2.6.3 Decentralized Glue

The Foundation is divided into multiple distributed components, i.e., intermediaries analogous to those attached to each SubEnv as in Figure 7.3 and loosely coupled with their local SubEnv, except that each of the SubEnvs may be a different system. There is no centralized component.

7.2.7 Choice of Heterogeneous Architecture

When a wide range of internal architectures is exhibited among the PCEs of interest, there is usually no obvious preference exhibited for centralized vs. decentralized glue, except as in any distributed system a decentralized approach will generally scale better; however, the extra software engineering effort of separately retrofitting a large number of existing local PCEs is more likely to be the limiting factor than a central bottleneck. Thus the number of local PCEs is expected to be small. By analogy to the discussion of Section 7.2.4, when scaling is not an issue, top-down global processes would generally be more amenable to a centralized Foundation, and bottom-up to a decentralized Foundation. But the case is not so compelling for heterogeneous as for homogeneous federation, there considering glue vs. peer-to-peer, since construction of a global process using diverse process modeling languages and paradigms is so complex as to overwhelm all other concerns.

This may be why Heimbigner proposes a third model for his process task server, a hybrid of top-down and bottom-up: *constructor* PCEs post pending tasks to the shared process repository in a generally bottom-up fashion, whereas it makes more sense for the *constrainer* PCEs, which remove disallowed tasks from among those posted, to be organized top-down. That is, constructors create newly instantiated tasks according to local process workflow, but constrainers may disallow some tasks according to global process constraints. A distinguished constructor, effectively part of the Foundation, might post multi-site tasks implementing a top-down global process.

The discussion in Section 7.4 considers a heterogeneous federation architecture where Oz plays the role of a constructor (and employs its own constraints prior to instantiating a task to post); other PCEs integrated into the same federation via ProcessWall could, of course, act as constrainers on the posted tasks as well as additional constructors.

7.3 The Oz Homogeneous PCE Federation

Oz is the only fully implemented homogeneous PCE federation that we know of.[2] Oz originally (versions 1.1.1 and earlier) followed the direct peer-to-peer architectural model, where the majority of the Foundation functionality was built into the process engine (as elaborated in [21]). Oz was later reimplemented (versions 1.2 and later), using a new process engine as moderated peer-to-peer — with the Foundation moderator separated out into a component invoked via generic "callbacks" from the process engine. We are primarily concerned with the later form of Oz in this chapter.

The overall choice of architecture for Oz follows the analysis given in Section 7.2.4. First, one of the major requirements for the Oz system is to support interoperability among autonomous, geographically distributed, and possibly pre-existing processes, e.g., the latter might have been designed for a single-site MARVEL environment; we developed a utility that mechanically upgrades an instantiated MARVEL environment to an Oz SubEnv. This requirement implies a bottom-up approach, which in turn suggests a peer-based architectural style. However, we have constructed both bottom-up (e.g., see [21], Appendix A) and top-down (e.g., see [125]) global processes for Oz.

[2]The Programming Systems Lab used a multi-site Oz environment to support all our day-to-day software development for about two years, since April 1995, although as of this writing we are transitioning to the OzWeb extension of Oz, which operates over the World Wide Web and supports a hypercode representation of product artifacts [122].

Second, Oz was developed (among other reasons) to interconnect instances of the Marvel framework. Since Marvel's client/server architecture corresponds to the centralized-process-control, decentralized-tool-execution local architecture, and Marvel's process engine provided no API and the source code was handy, it was natural to adopt a direct peer-to-peer approach. Later on, Oz's native process engine adapted from Marvel was replaced with the Amber process server [182], which provides an API (and "callback" interface), and hardwires neither centralized-process control nor decentralized-tool-execution. Amber itself was not modified at all to produce Oz multi-site functionality, as discussed in [121].

7.3.1 Marvel and Oz Overview

Everything described here about Marvel is also true for Oz unless stated otherwise.

Marvel [124] provides a rule-based process modeling language in which a rule generally corresponds to a process step, or task. Each rule specifies the task's name as it would appear in a user menu or agenda; typed parameters and bindings of local variables from the project objectbase; a condition or prerequisite that must be satisfied before initiating the activity to be performed during the task; the tool script with in, inout and out arguments for the activity [86]; and a set of effects, one of which asserts the actual results or consequences of completing the activity (some activities have more than one possible result). Built-in operations like `add`, `delete`, etc. are modeled as rules, and can be overloaded, e.g., to introduce type-specific conditions and effects on those operations; built-in operations can also be used as assertions in the effects of other rules.

Marvel enforces that rule conditions are satisfied, and automates the process via forward and backward chaining. When a user requests to perform a task whose condition is not currently satisfied, the process engine backward chains to attempt to execute other rules whose effects may lead to satisfying the condition; if all possibilities are exhausted, the user is informed that the chosen task cannot be enacted at this time. When a rule completes, its asserted effect may trigger forward chaining to automatically enact other rules whose conditions have become satisfied. Users usually control the process by selecting rules representing entry points into composite tasks consisting of one main rule and a small number of auxiliary rules (reached via chaining) for change propagation and automation of menial chores, but it is possible to define complete workflows as a single goal-driven backward chain or event-driven forward chain.

Marvel employs a client/server architecture [31]. Clients provide the user interface and execute tasks, usually by invoking external tools. The server context-switches among multiple clients, and includes the process engine, object management, and transaction management. Oz is essentially the same as Marvel, except that an Oz environment may consist of several servers, each with its own distinct process model, data schema, objectbase and tools [21]. Clients are always connected to one "local" server, and may also open and close connections to "remote" servers on demand. A server and its "local" clients constitute a SubEnv. The external view of the multi-site peer-to-peer Oz architecture is shown in Figure 7.6.

Oz servers communicate with each other mainly to establish and operate *alliances*, which involves (1) negotiation of *Treaties* — dynamically agreed-upon shared subprocesses that are automatically and incrementally added on to each affected local process on the fly when the Treaty is instituted (and automatically and incrementally removed when a site unilaterally revokes the Treaty); and (2) coordination of *Summits* — enactment of Treaty-defined process segments that involve data and/or local clients from multiple sites, with computation interleaved between shared and local computational models (i.e., the Treaty and the local processes). We stretch the International Alliance metaphor, since Treaties among sites precede and specify Summits rather than vice versa.

7.3.1.1 Treaties and Summits

The purpose of a Treaty is to establish a common subprocess. A Treaty consists primarily of a set of Oz rules. These rules define intentionally multi-site tasks, where the parameters are expected to be selected from multiple sites (i.e., distinct Oz objectbases chosen by a user via open connections

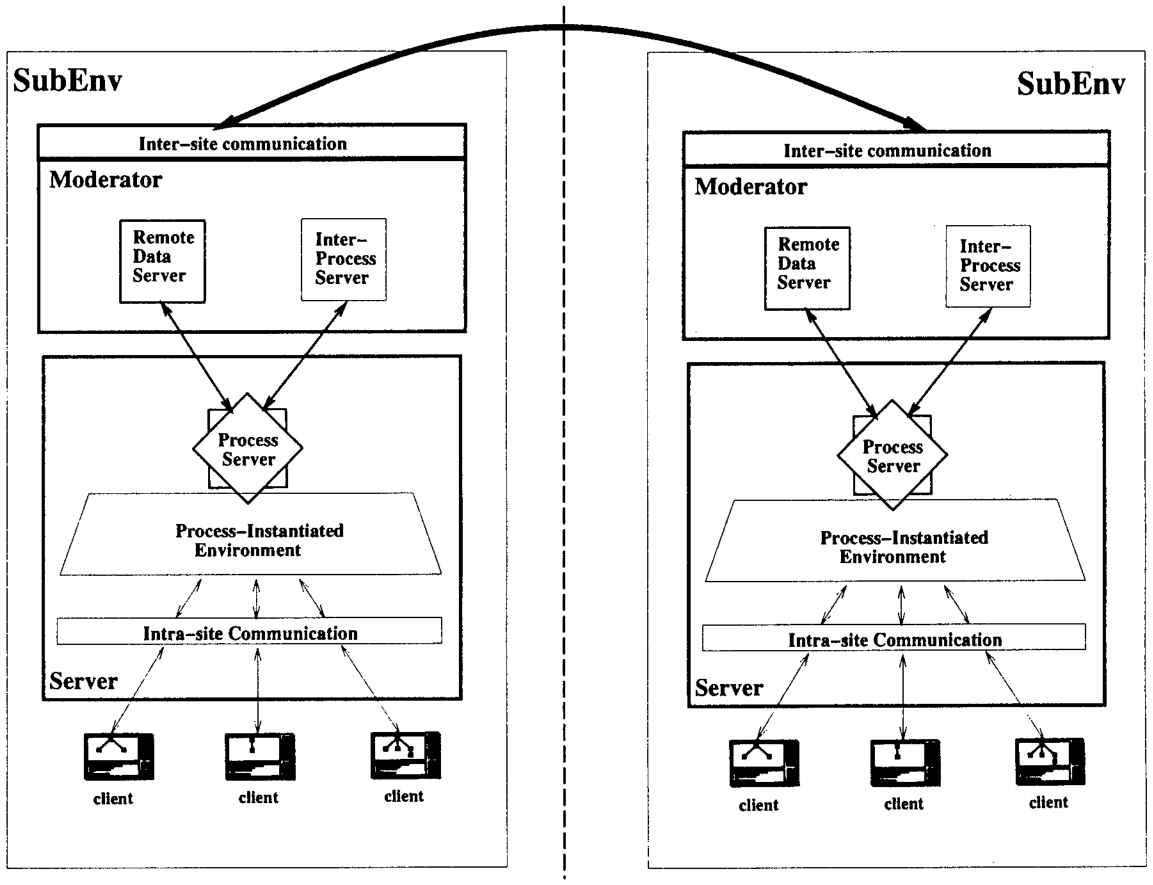


Figure 7.6: Oz External Architecture

to "remote" servers). Such tasks must be defined somewhere, in Oz's case within one of the participating SubEnvs. However, in general, Treaty rules are not an inherent component of any SubEnv's local process. Instead, the common subprocess is combined into each local participating process, in the sense that its tasks may be synchronized with other local tasks, depend on the outcome of their execution, and vice versa.

This is relatively easy to do with rules, the basis of Oz's process modeling formalism, since process enactment follows automatically determined forward and backward rule chains based on matching a predicate in one rule's condition to an assertion in another rule's effect [108]. It does not matter to the rule network construction algorithm whether the rules are included in the local process model or added later via a multi-site Treaty.

Treaties are defined pairwise between two Oz servers at a time, which allows local SubEnv administrators to form such agreements in a fully decentralized manner, without involving any global authority. Still, a Treaty among any number of sites can be created by forming all the relevant binary Treaties (and Oz provides commands to do this in one step, if the relevant administrator has appropriate privileges at each affected SubEnv). A Treaty between SubEnvs SE_1 and SE_2 over a subprocess SP is established when:

1. SE_1 issues an *export* operation of SP to SE_2. This operation assumes that SP already exists in SE_1 (either locally defined or imported from another SubEnv) and thus already integrated within its own local process. *export* also specifies execution privileges and general access control to the exported subprocess.

2. SE_2 issues an *import* command that fetches SP from SE_1 and tightly integrates it into its local process (the rules in a Treaty can be executed on purely local data, in which case they are not in any way distinguished from the local process).

In order to control execution privileges, i.e., which site(s) can initiate multi-site tasks (e.g., due to platform restrictions, security, etc.), both the *export* and *import* operation are parameterized with permissions to control on which site(s) those tasks can be executed and from which site(s) the relevant data can be fetched. That is, Treaties are not symmetric unless specified as such by both the *export* and *import* operations. To support decentralization, Treaties may be withdrawn unilaterally (except while multi-site tasks are actually being executed); note this requires dynamic Treaty validation, i.e., checking that none of the affected parties to the Treaty have revoked it. Thus, not only does the Treaty mechanism allow definition of decentralized multi-site processes, but the (meta-)process for establishing and maintaining Treaties is itself highly decentralized.

Summits are the process enactment counterpart of Treaty process models. When a multi-site rule (from some Treaty) is issued for enactment by a user client of its "local" Oz server, termed hereafter the *Summit coordinator*, that SubEnv performs the following main steps:

1. Verify that the corresponding Treaty is valid (i.e., it has not been retracted by one of the SubEnvs whose data was selected as parameters to the rule).

2. Evaluate the rule's condition to determine whether or not it is satisfied. This requires fetching all the parameters from their home sites and caching them locally.

3. If the condition is not satisfied, send to those participating SubEnvs whose data fail their condition predicates a request to issue *local pre-Summit* tasks, which involve local (hence private) process steps on local data with local tools determined via backward chaining from the requested multi-site rule.

4. Wait for all sites to return before continuing to the next phase. Note that if the original rule's condition is already completely satisfied, then the pre-Summit phase is null.

5. Execute the multi-site activity of the rule, usually but not necessarily involving data from multiple sites (it is possible that remote parameters appear only in the condition and/or an effect), and possibly multi-site tools (e.g., groupware).

6. Send to each participating SubEnv a request to update its own data affected by assertions of the rule's actual effect (determined according to the return code of the rule's activity).

7. Each such SubEnv issues corresponding *local post-Summit* tasks determining via forward chaining from the relevant assertions of the original rule's effect, again involving only local resources.

8. Wait for all affected sites to reply.

9. Enact further related Summits, if any, reached via forward chaining to other multi-site rules from the original Summit rule.

Thus, Summits alternate between execution of shared, global, and multi-site tasks, to execution of private, local and single-site tasks, and effectively enact multi-site processes with minimal inter-process dependencies beyond the explicitly defined shared subprocesses. Full details of Summits and Treaties are given in [28].

Treaties and Summits impose several requirements on the design of OZ's federated architecture. First, the tight process-level integration of an imported subprocess into the local process implies a strong coupling of the Foundation with the local PCE. The strict decentralization, even in the definition of the federated process (using Treaties), avoids the need for a global process state (except for the special configuration process, described in [26]), and further supports our choice of a peer-to-peer architecture according to the issues discussed in Section 7.2.4.

Finally, Summits do not require a global process controller, but do require functional extensions to local process engines to allow them to become Summit coordinators. Again, the peer-to-peer architecture is favored. However, none of the above aspects indicate any preference with respect to direct vs. moderated (peer-to-peer) approaches, assuming the process engine can be extended into a coordinator without internal code modifications — as is the case for the Amber process server but which would have been difficult for OZ's native process engine. Thus this decision is likely to be based on the lower-level architectural and implementation aspects of the local SubEnv — and, as noted above, we have tried both models and prefer the moderated approach on modularity and extensibility grounds.

Note that none of these concerns are specific to rules, as opposed to some other process paradigm, except in the sense that the rule network generation algorithm makes it trivial to tightly bind imported rules with the local process. Integration of imported Treaties into the local process for non-rule process paradigms is more complicated, but possible, as discussed in [28] and not addressed here.

7.3.2 Oz Architecture

The internal architecture of OZ is shown in Figure 7.7. We use the following graphical notations: squared boxes with the widest bold lines (e.g., the Server) represent operating system processes, or independent threads of control; squared boxes with lines with intermediate width (e.g., the Process component) represent top-level computational components that are part of the same operating system process as other components but are relatively independent from those components; squared boxes with narrow solid lines are computational subcomponents; dashed-line separators within subcomponents further modularize a (sub)component into its various functionalities; shaded rectangles within the above indicate "external" modules that extend the functionality of the basic component (as explained below); shaded ovals represent data repositories; and arrows represent data and/or control flow. The relative sizes of the various units are **not** intended to be meaningful.

OZ consists of three main runtime computational entities: the Environment Server (or simply, the *Server*), the *Connection Server*, and the *Client*. In addition, there are several utilities that convert the various project-specific definitions into an internal format that is understood and loaded by the server; they are of no concern in this chapter.

There are three kinds of interconnections: client-to-local-server, client-to-remote-server, and server-to-server. The first connection is "permanent", in the sense that its existence is essential

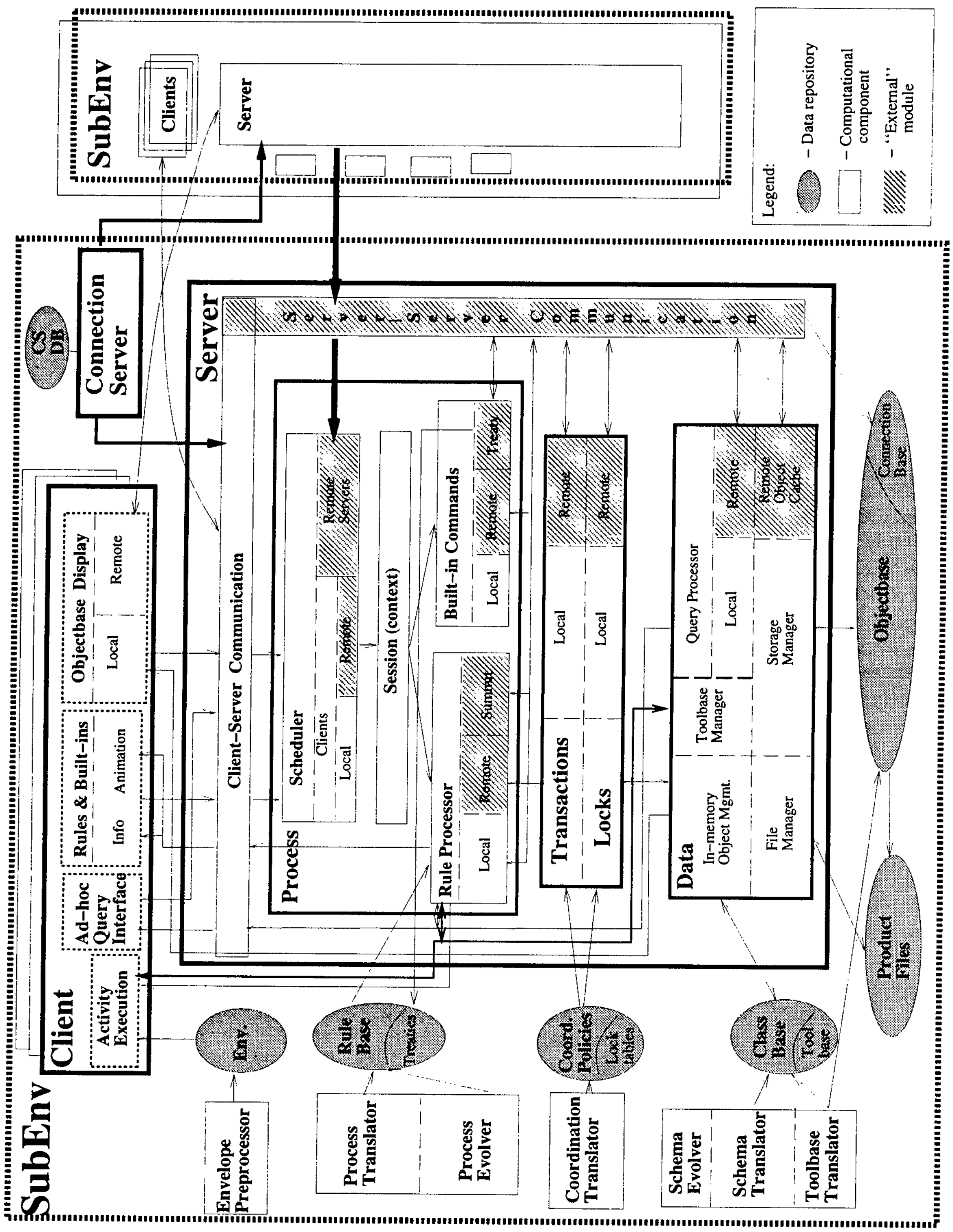


Figure 7.7: Oz Internal Architecture

for the operation of the client. That is, a client is assumed to always be connected to its local server, and when such a connection becomes disconnected (either voluntarily on demand or involuntarily due to some failure) the client normally shuts down and is removed from the local server's state.[3] In contrast, the two other connections can be regarded as "temporary", since they are optional, and can be dynamically reconnected and disconnected over the course of a user session without disrupting the local operation of a SubEnv. This is a necessary feature to fulfill the independent-operation requirement, particularly when the servers are spread arbitrarily over multiple administrative domains.

An Oz (multi-site) environment consists of a set of instantiated SubEnvs, and at any point in time none, some, or all SubEnvs may be *active*. A SubEnv is considered active if exactly one server is executing "on the environment", meaning that it has loaded the SubEnv's process, and the SubEnv's objectbase (containing persistent product data and process state) is under the control of the server's data management subsystem (described in [144]). Typically an active environment also has at least one active (i.e., executing) client connected to its server, because the server automatically shuts itself down when there are no more active clients (and is automatically started up on demand by the Connection Server, as will be explained shortly).

7.3.2.1 The Environment Server

The server consists of three major components: process, transaction and data managers, each of which can be separately tailored by a combination of two facilities: declarative definitions loaded from a file and "external" code modules. The process manager loads the process model (including portions obtained through Treaty *import*), the transaction manager is parameterized by lock tables and concurrency control policies, and the data manager loads the schema for the product data and process state (currently imported rule sets must employ subschemas compatible with the local schema, although some conversion is supported). All of these tailorings are stored in environment-specific files; see [184] for details. The conceptually "external" code is hardwired into Oz's data manager [4], reasonably independent and invoked through a callback interface in the case of the process manager, and completely independent and dynamically loaded for the transaction manager. We do not consider the distinction between "external" vs. intrinsic code further in this chapter, that is the subject of other chapters.

Process Manager

The process manager is the main component of the server. Its frontend subcomponent is the scheduler, which receives requests for service from three entities that correspond to the previously mentioned interconnections, namely local clients, remote clients, and remote servers. With few exceptions, notably to prevent deadlocks among mutual server communications, these requests are served on a first-come-first-served basis. The server is non-preemptive, i.e., it relinquishes control and context-switches only voluntarily.

The session layer encloses each interaction with a server in a context containing information that enables it to switch between and restore contexts. The context of locally executing tasks, including those that execute as part of a pre- or post- Summit, and the context of composite (multi-task) Summits, are represented in task data structures.

The rule processor consists of subcomponents for processing local tasks, local tasks spawned via pre-Summit or post-Summit processing from either local or remote Summits (denoted "Remote" in the figure), and Summit tasks. There are very few "system" built-in activities (notably parts of the configuration process), so the behavior of a particular instantiated SubEnv is mostly determined by the rule set that defines the process.

The built-in command processor handles all the kernel services that are available to every SubEnv. These include the primitive structural operations on the objectbase (e.g., add and copy object),

[3]An extension of this model, in which clients can be disconnected from their server and continue to operate independently to enact a process segment until reconnection, has been investigated separately to support mobile computing [210].

[4]Such code has been separated out in the later OzWeb.

several display options and image refresh commands, access control, and the various dynamic process loading and Treaty operations.

In the original direct peer-to-peer variant of Oz, all alliance support was hardwired. But in the newer moderated peer-to-peer versions, Summits, Treaties and related infrastructure has been culled out into the "external" code modules indicated in the figure.

Transaction Manager

All access to data is mediated by Oz's transaction manager. Due to the required decentralization, each transaction manager is inherently *local*, i.e., it is responsible only for its local objectbase. However, transaction managers attached to each server communicate among themselves to support concurrency control and failure recovery involving remote objects. Oz's transaction manager was developed separately and has been used independent from the rest of Oz. Further details are outside the scope of this chapter.

Data Manager

This component includes an in-memory object manager that provides uniform object-based access to data from any system component. Objects can be looked up in one of three ways: by structural navigation, by class membership, and by their object-identifier (OID). Structural and by-class searches are requested by the query processor to service navigational and associative queries, respectively, and by-OID lookup is used for several purposes, among them to support direct user selection of objects (mouse clicking in the objectbase display) as parameters to rules.

The second major subcomponent is the query processor. It supports a declarative query language interface, and is called from both the rule processor for embedded queries and directly from the user client for servicing ad hoc queries. Queries on remote objects are handled at this level, by invoking a server-to-server service.

The rest of data management consists of an untyped storage manager (implemented on top of the `gdbm` package) that stores the objectbase contents; a file manager that handles access to file attributes (file attributes are paths to files resident in the environment's "hidden file system"); and an object cache that holds transient copies of remote objects during Summits.

The data manager is tailored by the project-specific schema tied to the instantiated objectbase, including both class- and composition-hierarchies. As in the case of rules and the process manager, without a schema the data manager is useless since it cannot instantiate any objects.

7.3.2.2 The Client

There are three main clients, supporting XView, Motif and tty (command line) user interfaces, as well as several auxiliary clients with no user interface intended primarily for tool execution. Each client consists of four major subcomponents: (1) access to information about rules and built-in commands, (2) objectbase representation, (3) activity execution, and (4) an ad hoc query interface.

The two graphical user interface clients are (conceptually) multi-threaded, i.e., a single client can support multiple concurrent interactions with local or remote servers. This enables a user to run in parallel several (possibly long) activities from the same client. The command interface includes a process-specific menu and utilities for displaying rule definitions and the rule network interconnections, all of which are stored in the client's address space and can be dynamically refreshed when a new process is (re)loaded or a Treaty is formed. A dynamic rule-chaining animator shows the control flow of enacted tasks as they execute, both local and Summits.

The objectbase display maintains an "image" of *structural* information, i.e., parent/child and reference relationships, for browsing and for selecting arguments to activities. The contents of primitive and file attributes are transmitted only when needed. Users can select the `open-remote` command to display the objectbase images from other sites and subsequently select objects from multiple sites, allowing invocation of a Summit rule. The client maintains multiple simultaneous connections to the remote servers, and is able to direct requests to appropriate servers.

7.3.2.3 The Connection Server

The Connection Server's main responsibility is to (re)establish connections to a local server from local clients, remote clients, and remote servers. However, it does not participate in the actual interactions between those entities; it serves only as a mediator for "handshaking" purposes. In some cases, the destination server to which a request for a connection is made may not be active, in which case the Connection Server is capable of automatically (re)activating the dormant server. In other cases the desired server may be active but its address (host IP address and port number) might be unknown to the requesting entity, in which case the Connection Server sends that information to the requesting entity for further communication.

Unlike the Environment Server, the Connection Server is (conceptually) always active, since it is implemented as a daemon invokable via the Unix `inetd` mechanism. Thus, each configured host has its own (logical) Connection Server that supports all SubEnvs (of the same or different multi-site environments) that reside on that host.

7.4 Oz/ProcessWall: A Hypothetical Heterogeneous Federation

A heterogeneous federation is inherently more general than a homogeneous federation. Thus it is desirable to consider how a multi-site PCE like Oz might "fit" into a multi-PCE organized via a process state/task server, the only specific model we know of for heterogeneous federation of PCEs. Note the federation would presumably also include various non-Oz SubEnvs.

One approach is to drop the homogeneous Foundation entirely and employ only the heterogeneous Foundation for multi-site tasks. Then the homogeneous SubEnvs — i.e., homogeneous with respect to system but heterogeneous with respect to process model — would be treated as if they were unrelated local PCEs rather than part of an Oz multi-site PCE. Assuming that some component is added to interface with the federation glue, this should work trivially if they fulfill the requirement of independent operation — that is, that they do not depend on each other in any way to perform entirely local work. In other words, in principle we could have used ProcessWall to scale up MARVEL to support process interoperability across multiple teams, each with their own local MARVEL environment instance. But then the main advantage of a homogeneous federation is lost, namely the relative ease with which SubEnvs can call on each other to perform specific agreed-upon services within the identical (and thus mutually understood) process modeling and enactment paradigm.

An alternative approach is to allow individual (or all) SubEnvs of a homogeneous federation to participate in one or more heterogeneous federations, while retaining the higher level of intimacy afforded by the system-level homogeneity when (intentionally) interacting with other local components of the same system.

Note that SubEnvs that happen to participate in the same homogeneous federation may happen to employ each other's services indirectly through the heterogeneous federation, without necessarily any knowledge that they have more direct means of interaction. In fact, through this "backdoor" one might, in an unusual circumstance, inadvertently arrive in a situation where a SubEnv indirectly requests services from itself without realizing that its doing so — which could potentially happen in a homogeneous federation as well, although not in the Oz realization because the server checks for this degenerate case.

7.4.1 Issues

Assuming a centralized Foundation in the style of the ProcessWall state/task server, the main questions to answer are:

1. How would an Oz SubEnv post to the Foundation those tasks it has instantiated but not initiated, and (generally speaking) would like some other PCE to perform?

The local task descriptor must of course be converted to the Foundation's standard form. There are complications regarding representation of data arguments as part of the task specification, and later regarding data transfer when the task is enacted by some PCE participating in the federation. Even though the task may eventually be picked up by another Oz SubEnv, this cannot be assumed a priori (if it could, direct interaction through the homogeneous Foundation would almost certainly be more efficient).

2. How would the Foundation notify an Oz SubEnv of the completion of such a task?
The Foundation's relevant state must be converted to a form understood by Oz, and either automatically transmitted to Oz by the Foundation or explicitly requested by Oz, e.g., via periodic polling or some kind of rendezvous. When data arguments are modified during the task, changed data must either be submitted back to the Oz objectbase, or Oz must be notified of its whereabouts and have some means to retrieve the data.

3. How would the Foundation inform a particular Oz SubEnv that it should perform a specific posted task?
One approach involves some kind of scheduler or other entity that selects among enabled tasks for enactment, chooses the recipient SubEnv, and sends an appropriate request to that SubEnv. This Foundation-generated request model is compatible with Oz's current server-to-server communication mechanism. In contrast, a completely new interface would be needed to fit into a blackboard model that required each SubEnv to poll the Foundation for suitable enabled tasks. However, a hybrid might be achieved in the style of a broadcast message server like Field, where the SubEnv's *register* their interests in or abilities to perform certain kinds of tasks, perhaps by supplying a pattern that is matched by the Foundation against the enabled task specifications. The application of event subscription to workflow management system interoperability is suggested in [202].

4. How would an Oz SubEnv notify the Foundation of a task it had just completed?
There are two cases: the task was previously posted to the Foundation by the same or a different SubEnv, or it arose entirely inside the given SubEnv (and thus is supplied only as historic information). In the former case, the Foundation might have requested that this SubEnv perform the task, or alternatively the SubEnv might have selected the task from among those enabled (via either polling or registration). Note that generally it is necessary for the Foundation to prevent multiple SubEnvs from concurrently agreeing to perform the same posted task. Again, data transfer is of concern here in both directions.

7.4.2 Integrating Oz Tasks and ProcessWall Tasks

There are three different "levels" of task-like units supported by Oz: rule activities, full rules, and rule chains, any or all of which could be mapped to ProcessWall's notion of tasks. The answers to the questions above will be somewhat different depending on which Oz unit is chosen for the mapping.

Oz's lowest task-like "level" is an individual rule activity, i.e., invocation of a tool script (and thence external tool). Oz already defines a client/server protocol whereby user interface clients tell the server to apply a selected rule to a list of objects and literal arguments; once the condition is deemed satisfied, the server supplies the client with corresponding file pathname and primitive arguments, and directs it to invoke the tool script specified in the rule activity; the client forks the tool script, and the tool script or the tool(s) it invokes are assumed to directly modify the contents of file arguments; finally, the client returns to the server with the return code from the tool script, which selects among the possible rule effects, as well as (optional) assignments to output variables. The encapsulating rule (and its pending chain) then continues.

It is simple to construct a special Oz client that receives the same message from the server identifying tool script and arguments but does something different than the typical user client; in fact, we have already introduced numerous such clients (see [209, 56, 227, 148]). Then, to implement points 1 and 2, the new client would be inserted into the multi-PCE architecture between the Oz server and the central Foundation. This client would convert the activity information provided into the Foundation's task representation and forward it to the centralized Foundation. Later, after the activity has been completed, the Foundation would notify this special client, which would then respond to the Oz server like any of its other clients. The special client would be responsible for data traffic in both directions.

The same special Oz client (or alternatively a distinct special client) can be used to implement points 3 and 4. The Foundation sends the task to the Oz client for that client to execute itself using the same tool invocation facilities as any other Oz client, without involving the Oz server. When the task completes, the special Oz client returns the results to the Foundation, again without interaction with the Oz server. Note that some special communication facilities will be needed in the Oz client, if the same client is used for both purposes, to avoid deadlock when the client happens to be forwarding an activity to the Foundation to be performed by some other PCE at the same time that the Foundation is sending a request to the client.

The intermediate task "level" corresponds to entire Oz rules, with condition and effect(s) as well as activity. Oz servers already transmit rule definitions between themselves as part of Treaty negotiation, and transmit the parameters and bound variables of instantiated rules as part of Summit enactment. The newer variants of Oz introduce a protocol for transferring instantiated rules between client and server to support delegation to and selection from user and group agendas ("to do" lists) [121]. Rules with and without already satisfied conditions may appear in an agenda. These facilities might be combined and extended, again through a translating client interposed between Oz and the Foundation, to support all four points.

A complication: In the lowest level case the condition or prerequisite is already satisfied, by definition, prior to posting the activity, but this would not in general be true in the intermediate case. Analogously, handling the effect or consequence of the activity is the concern of only the originally posting PCE, but again this cannot be assumed in the intermediate case. The ProcessWall and Mentor task representations allow for predecessors and successors, but not all the constraints embodied in Oz conditions are concerned with checking simple predecessor relationships (e.g., the local variable bindings might find all objects that match a complex associative query and then the condition checks that at least one of those objects satisfies a complex logical clause), nor are all assertions made in Oz effects concerned with triggering successors (e.g., objects can be created and deleted, reference links formed and removed, etc. through invocation of built-in operations).

One could argue for a simplification, whereby Oz's postings to the Foundation are limited to those tasks whose conditions and effects are solely concerned with predecessor/successor relations that can be directly represented by the Foundation's state and/or task model. Although Oz's process modeling language tends to obscure such relationships from a human-readability standpoint, they are visible in the internal rule network compiled from the process model. Since the Foundation can only represent such relations, by definition any requests sent by the Foundation to Oz would so restrict the implicit conditions and effects.

A better approach might be to extend the Foundation's task representation, or develop some additional control channel, for transmitting the conditions and/or effects from the Oz SubEnv to the (potentially) foreign SubEnv for evaluation within its paradigm, and vice versa regarding communicating any prerequisites and consequences that might be supported by the foreign paradigm to an Oz SubEnv (and of course both issues come up between pairs of non-Oz SubEnvs as well). If Oz were configured as a multi-site homogeneous federation where some (or all) sites happened to also belong to a heterogeneous federation, pending tasks posted through the Foundation to another Oz SubEnv (in the same federation) could include their conditions and effects in some *opaque* data stream understood only by Oz servers.

So difficulties arise only when pending tasks posted through the Foundation involve non-Oz SubEnvs. Fortuitously, we have already shown fairly straightforward mappings from most of the major PCE paradigms, including Petri nets [183], task graphs [101], and grammars [127], into Oz rules, and reverse mappings are not inconceivable. And as previously noted in Section 7.1, Mentor supports translation from one notation into another, as does the process interchange format standardization effort.

However, the general case requires substantial translation capabilities regarding data formats, and predicates and operations over those formats. The "universal data model" problem is a well-known unresolved, probably unresolvable issue in database research [165]. It may be possible to address a special case of this problem with respect to PCEs, e.g., if we assume the main data arguments are files and all attributes that might be referred to regarding task prerequisites and consequences (rule conditions and effects in Oz) are standard file system appendages supported by most operating systems, such as owner, read/write timestamps, access permissions, etc., or encoded predecessor/successor relationships modeled directly by the Foundation.

The third task "level" in Oz is a rule chain, i.e., all the rules emanating from some user-selected (or Foundation-requested) rule through backward and/or forward chaining. This seems easiest to handle by iterating the intermediate case as the rule chain unfolds, in the Oz to Foundation case, or incrementally sending each member of a sequence of predecessor/successor tasks, in the Foundation to Oz case.

Note we have so far ignored the issue of specifying which PCE *user* should perform the work in the case of an interactive task. Oz version 1.1.1 included process modeling and enactment facilities, which could be revived, to delegate a rule activity to a specific user, or to one or all members of a user group [29]. Later versions of Oz can delegate an entire rule or (rest of a) rule chain to a specific user or any group member via "guidance chaining", a form of forward chaining where the next rule in the chain is placed in an agenda rather than immediately enacted [224]. How *another* PCE might designate a user to perform tasks originally instantiated by Oz is of course open-ended.

7.5 Contributions and Future Directions

The main contributions of this work are:

- The elaboration of requirements and architectures for homogeneous and heterogeneous federations of process-centered environments. Both the homogeneous and heterogeneous federation architectures we present are in line with a proposed distributed workflow reference model [229].

- The design and realization of a specific homogeneous federation architecture for Oz.

- A presentation of the issues that must be addressed to integrate Oz into a heterogeneous federation based on the ProcessWall process state/task server (or Mentor worklist/history manager) approach.

The obvious next step is to complete an experimental integration between Oz and the realization of a process state/task server, assuming one becomes available. [5] It would be desirable to also include in the heterogeneous federation at least one other PCE, besides Oz. Evaluation against the heterogeneous federation requirements should prove interesting.

Finally, we would like to introduce greater flexibility into Oz alliances, i.e., homogeneous federations, including navigation and search among related SubEnvs both within and across alliances, easy movement of user clients from one SubEnv server to another, and lighter weight composition and destruction of SubEnv alliances. Note the second point presumes support for arbitrary geographical dispersion *within* a SubEnv, not just *among* SubEnvs. Thus Oz user clients could not continue to assume a shared network file system for accessing objectbase file attributes and communication bandwidth may become a concern, issues already addressed to a limited extent for our Oz "low bandwidth clients" in [209].

[5] We have asked Dennis Heimbigner to give us a copy of ProcessWall many times over the past few years, to no avail.

Chapter 8

Flexible Process Enactment

Abstract

We present the design of a new rule-based process engine that generalizes previous systems to support process enforcement, automation, guidance, monitoring, delegation, planning, simulation, instrumentation and potentially other applications. Our approach is fully knowledge-based, tailored by knowledge regarding the *process assistance policies* to be supported as well as the *process definition.*

8.1 Introduction

A *process* is a set of steps for developing a software system. Different processes are needed for different projects, organizations and application domains. Thus, a software development environment intended to intelligently assist users in carrying out any of a wide range of processes should be knowledge-based. In particular, a *process model* represents the knowledge of the process steps, their prerequisites and consequences, and any synchronization among concurrent steps, defined in some *process modeling language* (PML). A process assistance system, generally known as a *process-centered environment* (PCE), includes an engine that interprets a given PML and helps the users in some way to carry out the specified process.

There are many different ways that a PCE could assist its users. It might *enforce* the process, by preventing any work that did not conform to the prerequisites of the process model. It might *automate* portions of the process, invoking tools at the right times with the right parameters. The PCE could *guide* users towards selecting the appropriate tools and/or their parameters to accomplish what needs to be done next, or it could *plan* a sequence of process steps to achieve the users' goals. It might allow one user to *delegate* one or more steps to another user, or it might *simulate* some or all of a process on sample data, as part of a tutorial for educating the user about the process (or the PCE). The actual process execution could be *instrumented*, to take measurements on relative frequency of choices among alternative process steps or typical numbers of iterations of cycles (*e.g.*, edit/compile/debug). Or the system might simply *monitor* each user's work, to be able to inform the users when they diverge from its knowledge of the process but without requiring conformance (the process model might be incomplete or wrong, and needs to be changed). There are probably several additional possibilities.

Existing PCEs generally support only a subset of these possibilities [47]. Our own MARVEL system supports enforcement and automation [124], but not the others mentioned above. We present a design for a fully knowledge-based process assistant, where the knowledge is not limited to the process model, but also incorporates the kind(s) of process aid required. Our new AMBER system will center on a generalized process interpretation engine, which can be parameterized so that individual environment instances support any subset of a wide range of process assistance options.

8.2 Motivating Example

The abstracted process model fragment shown in Figure 8.1(a) involves a code change during software development in the C programming language. The fragment starts when a header file is reserved and ends when the same header file is deposited, after the header file has passed code inspection <u>and</u> each of the affected source files has passed unit testing successfully. Each of the steps after reserve but before deposit can loop back to edit (loop not shown).

Alice and Bill are two programmers participating in this process. Say Alice wants to edit a header file; a PCE that enforces the process won't let her edit until she has reserved the file. Process enforcement ensures that prerequisites defined in the process model are strictly followed. In contrast, process automation doesn't prevent users from doing anything, but instead opportunistically undertakes simple activities on its own so users need not be bothered with them. For example, after Alice modifies the header file, a PCE might try to re-compile all the affected source files.

Guidance may be preferable to automation when a PCE doesn't have sufficient knowledge to perform process steps by itself. Consider the case that the header file Alice wants to reserve is already reserved by Bill. It is usually inappropriate for a PCE to automatically deposit the file for Bill, since Bill might not be done with his changes yet. Instead of doing something inappropriate, or nothing at all, a PCE might insert the deposit step into Bill's agenda ("to-do" list), but the decision as to when to select this agenda item is up to Bill. This particular case can be seen as process delegation, since Alice indirectly asks Bill to deposit a file she wants by requesting to edit it, but a PCE could also insert agenda items on behalf of the same user (*e.g.*, to later compile a file that he or she had edited).

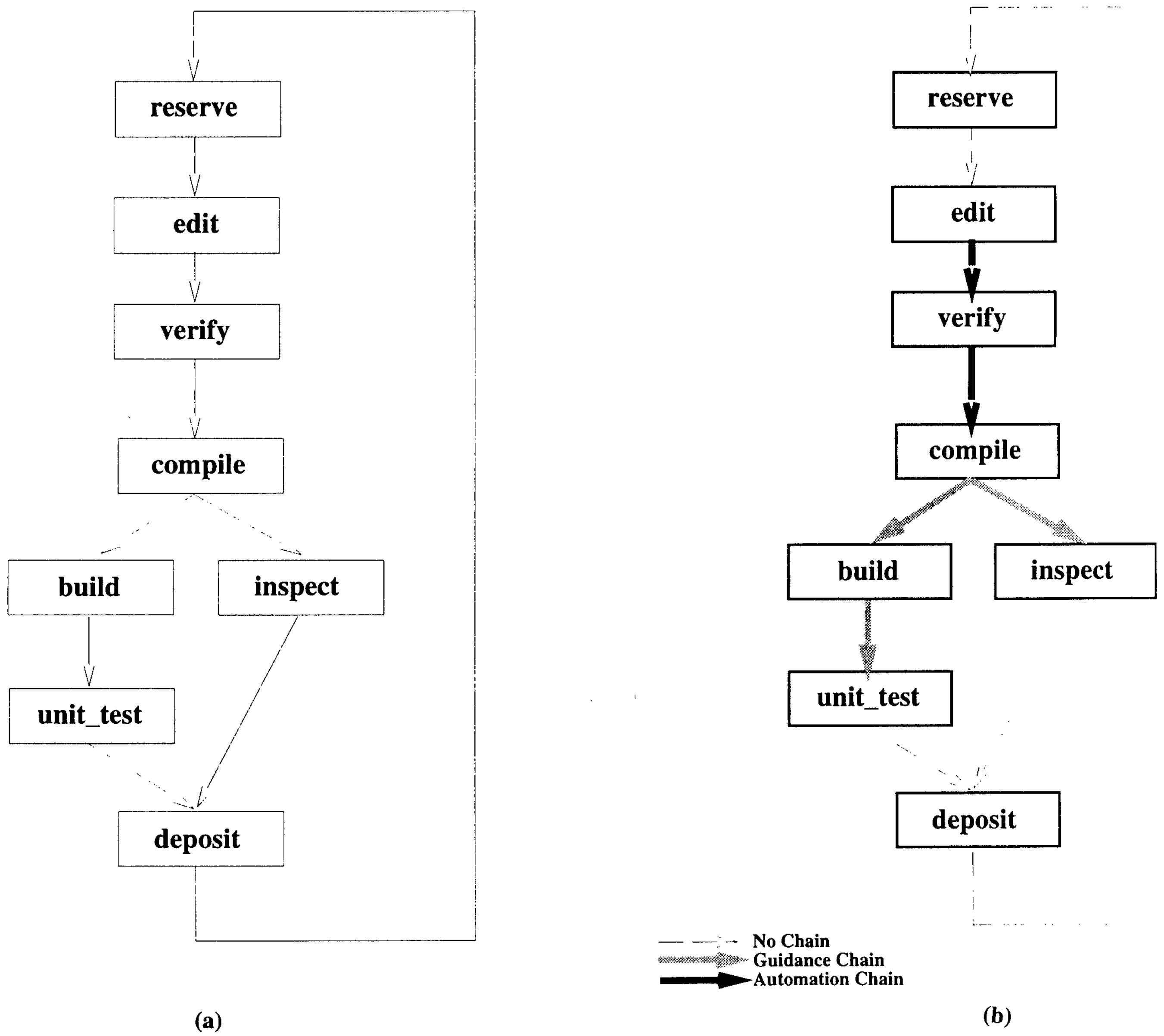


Figure 8.1: Simple Process Fragment

When process enforcement is viewed as an undesirable constraint on individual freedom, monitoring offers a non-intrusive alternative by letting the user do whatever he or she wants. For example, if Alice is not willing to wait for Bill to release the header file before beginning her edits, a PCE might turn off process enforcement and switch to process monitoring (probably only with respect to Alice; the other users need not be affected). Then it would be possible to let Alice go ahead and make her change on a copy of the file, but the PCE records the fact that the normal procedure is broken (maybe the process should be changed to permit parallel reservations of the same file) and send a note to Bill to let him know what's going on.

When Alice is unsure whether to edit the header file or not, she may appreciate a PCE that can do process planning, to find out what are the likely consequences of her proposed change, such as what other code would have to be re-compiled. And if Alice is unfamiliar with the process, because she is new or the process has recently changed substantially, she might first (or instead) run a process simulation. The PCE would go through the selected portion of the process model considering the data indicated by Alice, showing her how the process works, but without generating permanent side-effects on the data.

Finally, if Alice is frantically trying to meet a deadline, she might want to know how long it usually takes to re-build and test the affected source files after a change to that header file. A PCE supporting process instrumentation should be able to offer such measures. If the build and test steps could be prolonged, she might forego her change.

In all of the above situations the <u>same</u> process model (*i.e.*, the process model depicted in Figure 8.1(a)) is followed, only the interpretation of it is different — as needed to support different process assistance modes. Since the full range of plausible modes is unknown, a PCE kernel that supports only basic process modeling *mechanisms* and leaves process assistance *policies* to its instances would minimize the effort to implement the desired set of modes for a particular project (and may also make it feasible to add or delete some modes while the project is in progress).

8.3 Knowledge-Based Assistance

We are interested in two kinds of knowledge: (1) the knowledge of the specific process; and (2) the knowledge of the kind of assistance to provide. All process-centered environments support (1), by definition. We are primarily concerned here with (2). Further, although process models may be expressed in a number of different paradigms — Petri nets and grammars are also popular (see [132]) — we limit the discussion to *rules.*

Let's say, generally, that a rule consists of three main parts: precondition, action, and postcondition. The precondition specifies any prerequisites to the action, and the postcondition specifies any consequences. Many rule notations (logic programming, production systems, database triggers, *etc.*) use two parts, combining the action with either the precondition or the postcondition, but the extension from our discussion of three-part rules to either form of two-part rules is straightforward. Rule-based PMLs typically also include a name and formal parameters, so that a rule can be explicitly invoked to execute the named process step on the given actual parameters.

The *rule execution model* consists of (at least) five stages: selection of a rule, binding of its parameters and local variables, evaluation of the precondition, performance of the action, and assertion of the postcondition. To support multiple process assistance modes, no specific policy should be hard-wired into the kernel as to how to behave if any of these steps succeeds or fails (although there may be defaults that can be overridden). For example, in a PCE that only enforces the process, if the precondition evaluates to false, no further stages can be executed; but if the PCE supports automation as well as enforcement, then backward chaining could be initiated to attempt to satisfy the precondition (a PCE might support only partial automation, *i.e.*, only backward or only forward chaining, not necessarily both). In contrast, a PCE that monitors rather than enforces the process might simply record that the precondition failed (perhaps after automated backward chaining) and go on to the action performance stage.

It should also be possible to skip a stage entirely, usually the action, or to apply the stages to private copies of data maintained throughout a chain or for an entire user session. Then during planning, a sequence of rules might be produced from a goal (backward chaining) or an event (forward chaining), without performing actions and applying postconditions to private data kept only for the duration of the chain. A simulation might or might not skip the actions depending on the purpose of the simulation, *e.g.*, training *vs.* analysis of dynamic behavior, and employ the same private data across multiple chains.

The *rule network* is an internal representation of the rule base. In theory, this could be omitted and the entire rule base re-parsed over and over again as needed, but this is inefficient. What we have in mind when we choose the term "network" is a directed, potentially cyclic graph, where each vertex is labeled by a rule, and each edge is labeled by a predicate in the precondition of one rule and the postcondition of another (or the same) rule. Each edge may be annotated with one or more directives concerning the applicable process assistance modes. Other formats may also be suitable, depending on the chaining algorithms adopted.

A *rule instance* is the internal representation of an individual rule used during rule execution. That is, one or more of the stages of the rule execution model have been completed, but not all of them. Rule instances can be strung together into agendas, and must be (potentially) persistent so that such contexts can be remembered between user sessions. The most useful scenario for manipulating instantiated rules is probably after selection, parameter binding and precondition evaluation, but before action performance and postcondition assertion; such rules have been "enabled" but not yet "fired". Another practical possibility is after selection and binding, but before precondition evaluation, since a process step might reasonably appear in a user's agenda long before it is ready to go. But, for generality, any prefix of the stage sequence should be supported. There should also be some means to "clone" rule instances to appear in multiple agendas associated according to the process assistance policies — each with a named user, a group, a role such as programmer vs. technical writer, and so on. Alternatively, the same rule instance could be referred to from several agendas, and when executed or deleted by any user, it disappears from all. In any case, there must be mechanisms through which a user executing an instantiated rule can directly or indirectly (through chaining as discussed below) generate new instantiated rules, to add to his or her own agenda or delegate process steps to other users.

The *forward chaining algorithm* should supply generic means whereby applying the execution model on one rule can result in applying the execution model to any or all of the rules reached by rule network edges emanating from the postcondition of the original rule. The algorithm should be parameterizable to treat edges with different annotations differently, *e.g.*, to consider all the automation edges first, and then turn to guidance edges. Automation edges might result in attempting to complete all five stages for the relevant rules, whereas guidance edges might only place (partially completed) rule instances into agendas. A PCE that supports only process monitoring, as opposed to process enforcement, might check whether or not the sequence of rules actually chosen by the user matched an internally simulated forward chain (note that a chain may be partially ordered or cyclic, not just a strict sequence).

The *backward chaining algorithm* should include generic means whereby when one applies the execution model to one rule, the execution model can be applied to one or more of the rules reached by edges emanating from the precondition of the original rule. Again, the algorithm should be parameterizable to consider only edges with specific annotations, and whether only one or all accordingly annotated edges should be considered. And also again, it must be possible to treat edges with different annotations differently. Finally, the policy needs to specify whether the backward chaining (with respect to a specific annotation) should halt after one or a specified depth/number of AND/OR branches is attempted (*e.g.*, placing an instantiated rule in an agenda for guidance), after the original precondition becomes satisfied (*e.g.*, when automation is supported), or after trying all possibilities. The first two cases seem more practical than the latter, but a fully knowledge-based process assistance mechanism should allow for the construction of unforeseen process assistance modes. It should also be possible to turn off backward chaining and/or forward chaining entirely in a particular PCE instance.

To achieve the various parameterizations, the rule execution model and the chaining algorithms should incorporate "hooks" for inserting functions specific to the PCE instance, for example, to decide whether to skip a stage or deal with a failure, how to order among rule network edges and traverse the network, and also to implement instrumentation, so that various measures can be obtained by on- or off-line analysis of the history log. One application might be to determine worst case and average case lengths of time for a designated action to complete, with respect to particular data or any data. Another would be to track how often a particular rule's precondition is satisfied, on particular data or on any data. And information about the origination and contents of chains should be accessible, to determine how often execution of a certain rule leads to consideration of another specific rule, or of any rule(s).

Both chaining algorithms depend on a *dynamic data binding mechanism*, whereby parameters are "passed" from one instantiated rule in a backward or forward chain to another. This issue is subtle and has already been reported in [108].

8.4 MARVEL Background

The knowledge of the process (or workflow) is represented in a rule-based process modeling language. Each process step is encapsulated in a *rule* with a name and typed formal parameters. Each rule is composed of a *condition* (the precondition of the generic rule-based PCE above), an optional *activity* (the action), and a set of *effects* (collectively, the postcondition). The condition has two subparts: bindings, to query the objectbase to gather local variables (*e.g.*, included ".h" files when compiling a ".c" file), and properties, which must evaluate to true prior to invocation of the activity. The activity names a tool envelope and specifies its inputs and outputs. *Envelopes* are stylized shell scripts, which permit conventional file-oriented tools to be integrated into an environment instance without source modifications or recompilation [86]. Each envelope implicitly returns a code that uniquely selects the specific effect from among those given, to assert the actual result of the tool on the objectbase (*e.g.*, returning 0 for a successful compilation selects the first effect, returning 1 for detection of syntax errors maps to the second, *etc.*). A separate effect is necessary since an external off-the-shelf tool can directly modify files referred to in the objectbase, through the shared file system, but cannot directly manipulate the internal objectbase. If there is no activity, then there is only one effect.

When a user enters a command, MARVEL selects the matching rule. The condition of a rule is *satisfied* if its properties evaluate to true for every binding of each local variable (in the case of `forall` bindings) or to at least one binding of each local variable (for `exists` bindings), considering the actual parameters supplied to the command. If the condition of a selected rule is not satisfied, backward chaining is attempted. If the condition is already satisfied or becomes satisfied during the backward chaining, the activity is initiated; otherwise the user is informed that the process step cannot be carried out at this time.

After the activity finishes, the appropriate effect is asserted. This triggers forward chaining to any rules whose conditions become satisfied, which are executed in an implementation-specific order obeying the directives below. The asserted effects of these rules may in turn satisfy the conditions of other rules, and so on. Eventually no further conditions become satisfied and forward chaining terminates. Chains affiliated with different users are interleaved at the points where activities are executed, and new commands and activity returns are scheduled in FCFS order. To support such multi-tasking, MARVEL keeps a *stack* of rule instances for each chain, and context-switches among the stacks. (Like most rule-based PCEs, the MARVEL process engine is centralized, with a client/server architecture [31]; we describe a multi-server decentralized extension in [27].)

Forward and backward chaining through condition and effect predicates is controlled by chaining directives. Forward chaining is optional, and can be "turned off" wholesale or explicitly restricted through `no_forward` or `no_chain` directives on individual automation predicates in the effect. Backward chaining can also be "turned off", or restricted by `no_backward` or `no_chain` directives on individual predicates in the condition. Possible chains are compiled into a network when the process model is installed, with edges annotated as to chaining type (*e.g.*, `automation`). Both backward

```
arch [?m:MODULE]:
 (and (forall CFILE ?c
              suchthat (member [?m.cfiles ?c]))
      (forall MODULE ?ch
              suchthat (member [?m.modules ?ch]))
      (exists AFILE ?a
              suchthat (linkto [?m.afiles ?a])))
 :
 (and no_chain (?m.archive_status = NotArchived)
      no_forward (?m.compile_status = Compiled)
      no_forward (?c.archive_status = Archived)
      no_forward (?ch.archive_status = Archived))

 { ARCHIVER mass_update ?m ?a.history }

 (and (?m.archive_status = Archived)
      no_chain (?m.time_stamp = CurrentTime)
      (?a.archive_status = Archived));
 (?m.archive_status = NotArchived);
```

Figure 8.2: A Rule From C/MARVEL

and forward automation chaining proceed in a depth-first order with respect to the rule network.

Figure 8.2 shows a (simplified) rule from the process we used in MARVEL's own development (which consists of 40 classes, 184 rules and 46 tool envelopes). This `arch` rule checks whether its `MODULE` parameter has already been archived and permits backward chaining attempts to compile the `MODULE` parameter and/or to archive any of the `CFILE` components or nested `MODULE`s. However, separately executing an `arch` rule (overloaded, only the one for `MODULE`s is shown) on one of these components does not automatically forward chain to trigger this rule on the enclosing `MODULE`, as this is prevented by the `no_forward` directive. When one of `arch`'s effects is asserted, forward chaining is initiated. For example, whenever `arch`'s second effect is asserted, then any other rules whose conditions are satisfied by setting the status of a related `AFILE` to `NotArchived` will be automatically executed.

8.5 Amber Design

We plan to generalize MARVEL's rule execution model by replacing the hard-wired operations governing each stage with hooks for *callback* functions, defined separately for distinct process engine instances (that is, versions of the engine supporting different sets of process assistance modes). Such callback functions would be invoked immediately before and immediately after each stage, and return a boolean value. They would determine whether or not to skip a stage, and decide what to do when the stage succeeds or fails (e.g., continue or terminate individual rule execution, or rule chaining, depending on the stage). The chaining annotation for each edge in the rule network would become a value of an arbitrary *enumerated* type defined by a process engine instance, not just one of the automation directives.

A *state* and an *origin* field would be added to the rule instance data structure, with each possible value of *state* corresponding to one of the stages and *origin* recording how the instance was originally generated, such as by explicit user construction or *via* one (or more) of the enumerated chaining annotations. These fields together with the edge annotation would parameterize the dispatch to the callback functions. Thus callbacks would permit variation in both the interpretation of the rule notation and the semantics of chaining among rules. It is critical to understand that callback functions would be defined per stage and per annotation, but not per rule, so the callback functions would be entirely localized to the process engine and the number of functions that need to be implemented for each process engine instance would be reasonably small — and defaults would be provided that duplicate MARVEL's original functionality.

To realize process delegation, a rule instance would also contain a generic pointer ("void *" in the C programming language) to a *context* data structure defined in a process engine instance. For example, if the context represents a user session, the rule instance might be executed on behalf of that user and its activity will be performed by that user; if the context is an agenda, or a list of agendas, the rule instance might be entered into (or deleted from) each of these agendas. New routines would read/write the rule instance data structures from/to *persistent* storage, necessary to save an agenda between user sessions.

To allow both breadth-first and depth-first chaining order, a *tree* of rule instances per task would replace the rule instance stack. Each rule instance would contain a pointer to the "parent" instance that was responsible for creating it, as well as two lists of "children" created by forward and backward chaining. Some way of distinguishing among "children" would be needed, since both directions of chaining may lead from a single rule instance. For example, a rule instance created by backward automation chaining might trigger only additional backward automation chaining, or both further backward automation chaining and forward guidance chaining (see next section). These chaining dependencies between rule instances would comprise the task tree, which would be created when a user invokes a rule, expanded and shrunk during chaining, and deleted after all the rule instances finish their execution.

To determine ordering among the "children" of a given instantiated rule and across instantiated rules, the kernel would maintains a set of *queues* of rule instances. These would be managed by the callback functions associated with conditions and effects, to schedule the execution at the different priorities determined by process engine instances. The rule instances in queues of lower priority would not be executed until the queues of higher priority were empty. For example, rule instances created by automation chaining might be placed in a queue having higher priority than the queue where those generated by guidance chaining were placed, or vice versa, depending on the desired process assistance policies. Within a queue, the rule instances would be executed according to the usual FIFO scheme.

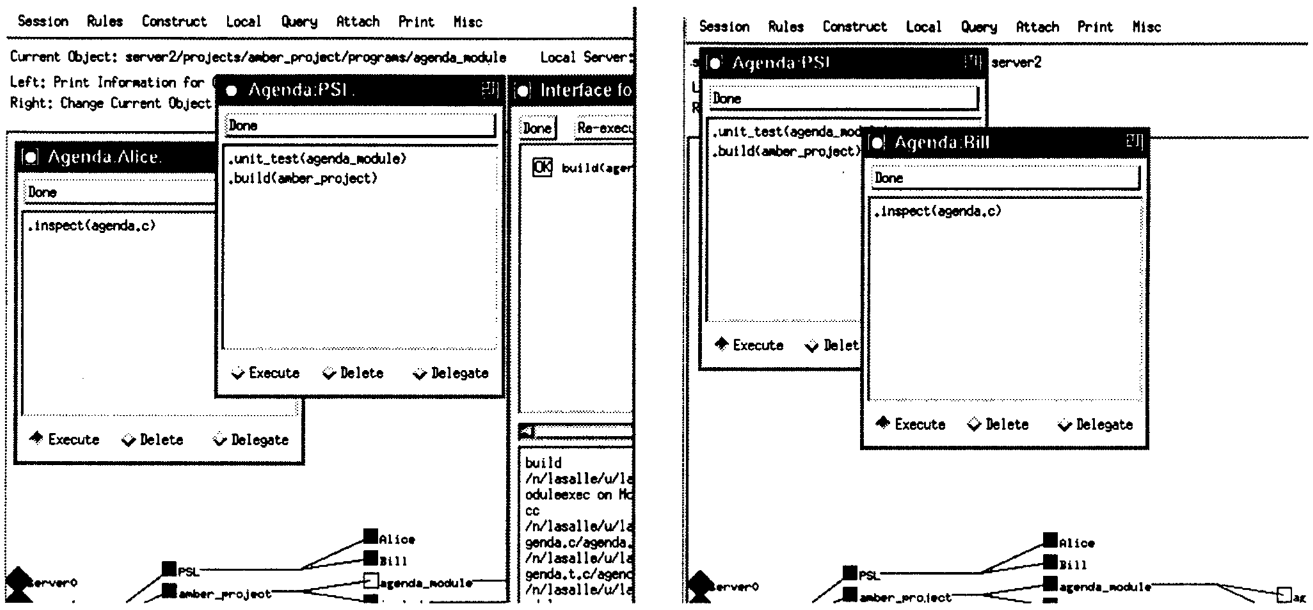
The next section describes the guidance and delegation facilities that have already been implemented (except as noted) in a preliminary version of AMBER; the following section describes our ideas for how we might implement some other process assistance modes.

8.6 Guidance and Delegation

An agenda is a persistent data structure that can be attached to and detached from a user session dynamically by the user. When attached, a special agenda window displays the list of rule instances, each of which has been instantiated with data parameters after at least the binding and possibly the precondition stages of the rule execution model (see above). Each agenda entry can be executed, deleted or delegated to other agendas by the user. If we stored agendas as instances of a new built-in `AGENDA` class (and its project-specific subclasses) in the objectbase, then they could be browsed and queried using MARVEL's existing facilities, but for now they are saved in a separate file.

The process engine automatically inserts instantiated rules into agendas through *guidance chaining*. A new **guidance** annotation was added to MARVEL's PML to mark selected condition and effect predicates. Optional arguments to **guidance** annotations associate such predicates with one or more named agendas, so that the process engine knows to which agendas to add the rule instances generated through guidance chaining from that predicate; the default is the agenda of the user who triggers the guidance chaining.

Guidance chaining begins like automation chaining: when a rule is reached in the rule network (after the completion of some original rule selected by a user command), dynamic data binding is applied to generate a rule instance, and the condition is evaluated. But then instead of attempting to complete the action and postcondition stages of the rule execution model, the rule instance is



```
Current Object: server2/projects/amber_project/programs/agenda_module
```

```
.inspect(agenda.c)
```

```
.unit_test(agenda_module)
.build(amber_project)
```

```
build(age
```

```
build
/n/lasalle/u/l
odulesexec on Mc
cc
/n/lasalle/u/l
genda.c/agenda.
/n/lasalle/u/l
genda.t.c/agen
/n/lasalle/u/l
```

Figure 8.3: After Alice edited "agenda.c" and built "agenda_module".

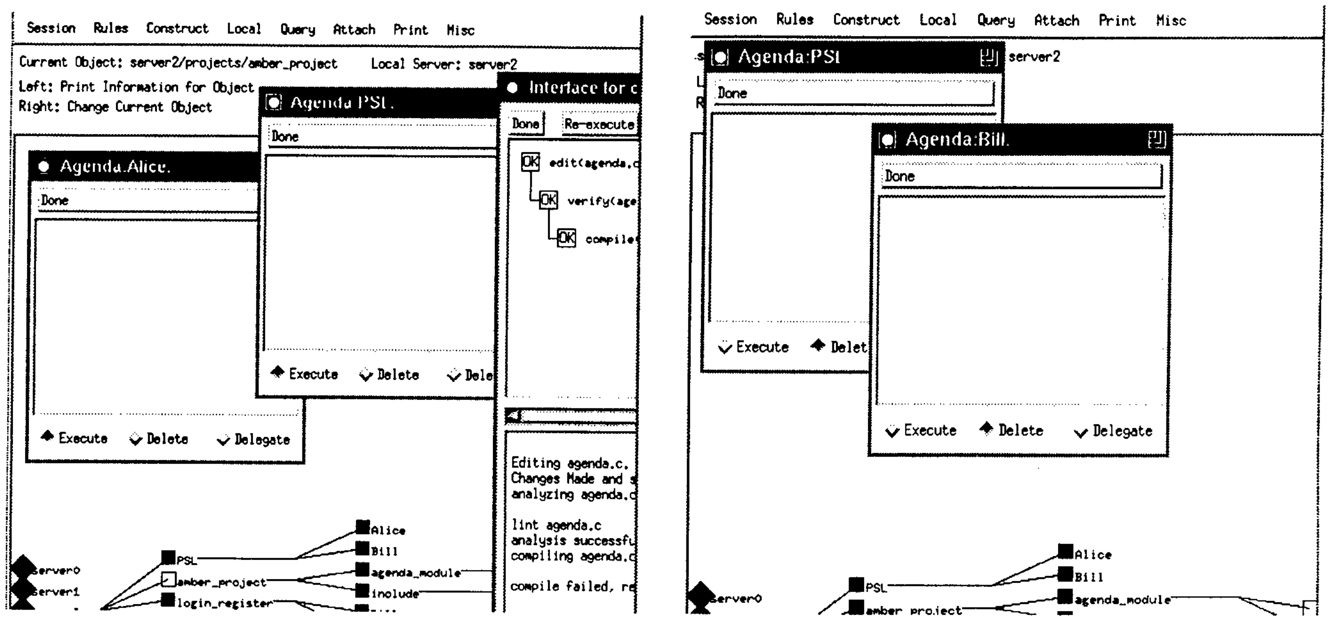
inserted into one or more agendas after the precondition stage (if the precondition is satisfied). Thus the choice of when and whether to actually execute the rule instance is up to the user(s). Guidance chaining is applied only after automation chaining emanating from the same original rule has finished, that is, automation chaining has higher priority.

The PCE can keep the agendas up-to-date, containing only rule instances with satisfied conditions, through *negative chaining.* Unlike *positive chaining* (*e.g.*, automation, guidance), which generate new instantiated rules, negative chaining finds the existing rule instances whose conditions previously evaluated to true but might possibly evaluate to false due to the just-completed execution of another rule. The process engine will re-evaluate the conditions of all such rules. If any condition is no longer satisfied, that entry would be removed from its agenda(s).

The cost of negative chaining grows with the number and size of agendas, so it should be possible to "turn off" negative chaining entirely and/or focus it to affect only the agenda(s) of the same user (who triggered the negative chaining). The penalty of such relaxation is that the process engine can no longer guarantee that the conditions of existing agenda entries are always satisfied, so when a user chooses to execute such an instantiated rule, its condition must be re-evaluated at that time — and if it should prove unsatisfied (or unsatisfiable through automation backward chaining), that process step cannot be completed (assuming process enforcement). A refresh mechanism could periodically re-evaluate the conditions of rule instances in agendas based on policies specified by the relevant user(s), such as after certain number of rules have been executed or a certain period of time has elapsed.

Figure 8.1(b) depicts how we model the abstract process fragment in Figure 8.1 (a), with edges annotated to specify the guidance capabilities described here. Figures 8.3 and 8.4 give snapshots of the screens for Alice (left) and Bill (right) when they follow this process. Each figure shows both Alice and Bill with two agendas, their user agendas ("Alice" and "Bill" respectively) and a shared group agenda ("PSL").

After Alice edited a CFILE "agenda.c" and made a change, the `verify` and `compile` rules were automatically invoked (`verify` calls the `lint` tool, which checks for certain common classes of errors that are not detected by the weakly-typed C compiler). After the compilation succeeded, the process engine evaluated the `inspect` rule on "agenda.c" and the `build` rule on the "agenda_module", and inserted appropriate rule instances into Alice's and Bill's user agendas. Figure 8.3 shows what happened after Alice executed the `build` rule from her user agenda: After the "agenda_module"



```
Current Object: server2/projects/amber_project     Local Server: server2
Left: Print Information for Object
Right: Change Current Object
```

```
Editing agenda.c.
Changes Made and s
analyzing agenda.c

lint agenda.c
analysis successfu
compiling agenda.c

compile failed, re
```

Figure 8.4: After Alice edited "agenda.c" again.

was successfully built, the system guidance chained to the `unit_test` rule on "agenda_module" and then the `build` rule on the "amber" project. Both of these rule instances were then inserted into the "PSL" group agenda. The "Alice" user agenda was not affected. Then, Alice edited "agenda.c", and the system automation chained to the `verify` and `compile` rules again, but this time the compilation failed — resulting in negative chaining to remove the `inspect` rule from Alice's and Bill's user agendas as well as the `build` and `unit_test` rules from the "PSL" group agenda, as depicted in Figure 8.4.

8.7 Planning, Simulation, Monitoring and Instrumentation

Process planning generalizes process guidance: instead of looking at only the next step in a chain and placing it into an agenda, planning continues through an arbitrary number of steps until the "chain" is done. We expect to realize process planning by caching a copy of each data item accessed during planning, and applying automation chaining to the private copies. Since activities are not executed during process planning, a plan might consider every effect in a full search, or only one "most likely" path.

Like process planning, we are thinking that process simulation would automatically "fire" rules on cached copies of data. However, here a "standard" effect (not necessarily the most frequent one) might be specified for each rule by another annotation in the process model, probably reverting to a second choice after several repetitions when there is a cycle. Further, the PCE could inform the user of what is going on as the process "executes", using the animation facilities mentioned below, as opposed to generating a list of process steps for later execution.

Process monitoring involves a fairly simple change: When the condition of a rule instance cannot be satisfied by backward chaining, instead of terminating the rule execution model at that point, the process engine could continue to the next stage (executing the activity) but warn the user and record the divergence from the predefined process model in a log file.

Process instrumentation would normally be realized by attaching functions to the rule execution model and chaining algorithms. But MARVEL conveniently provides a simpler alternative: when the user is performing a process fragment, that fragment is "animated" on a graphical display, with icons for each activity executed and special arrows between icons indicating backward *vs.* forward chaining. So instead of modifying the process engine in several places, we are instrumenting the animation module to capture all the events it already receives.

8.8 Related Work

Merlin [179] offers process guidance by providing *working contexts* for each user, which display the set of software objects, their inter-dependencies and the possible next process steps to be performed on all these objects — based on the user's specific role and the task at hand. The contents of a user's working context can be updated dynamically due to the results of others' work through backward chaining on special backward chaining rules, whose notation is different from the separate rule base used by forward chaining to achieve automation. Process WEAVER [67] uses a Petri-net process modeling formalism centered on *cooperative procedures*, which specify the ordering of *activities* (process steps) and the conditions that trigger the move from one activity to another. Each user has an *agenda* that contains all the *Workcontexts* encapsulating activities assigned or delegated to him that form a "to-do" list. Process guidance and delegation is achieved by defining an activity in a cooperative procedure to put Workcontexts into a user agenda, and letting a user dynamically construct and delegate Workcontexts to others.

EPOS [151] uses planning in its implementation of process automation. Its primitive *tasks* are essentially planning system-style rules, but with additional input/output *constraints* that achieve parameter passing among tasks. A planner refines high-level tasks into partially ordered sequences of subtasks, ultimately into individual rules. The resulting *hierarchical plan* specifies the control flow that the EPOS execution manager then assists users in carrying out. This approach breaks down when a (sub)task fails to achieve its goal, but an incremental replanning mechanism has been proposed. Grapple [111] plans software processes off-line, for manual execution by process participants. In principle, it could also monitor in-progress processes as input to plan recognition.

StateMate [112] models the traditional "who, what, where, when and how" of a process and simulates the dynamic or behavioral aspect of a process through *state transitions* in a finite state automaton. Transitions are labeled with a *trigger* (including an event expression and a condition expression) and/or a set of *actions* (*e.g.*, generate an event, set a condition, perform a calculation, *etc.*). When the trigger requirements are met (the event expression occurs while the condition expression is true), the state transition is taken and the actions are performed, moving the process to the next state. However, there are no means to actually execute software development tools or to store product data, so the process execution can only effect a simulation. Articulator [159] simulates a process by symbolically "executing" tasks according to the *task precedence structure* specified in the process model. The simulation proceeds as long as task conditions are satisfied at each step; otherwise, the simulated process stops, reports the problem, and waits for input or command from the simulation user.

The Articulator process engine is also instrumented to monitor any deviations from the specified process (during a non-symbolic, actual process execution), and collect process data including the ordering and duration of activities, tools invoked, users who have worked on different tasks, *etc.* Such information is abstracted and fed back through a sophisticated meta-process to improve the process model. Amadeus [203] instruments a process model with *agents* written in an interactive script language. When triggered by user-specifiable process events, object state changes and/or calendar time abstractions, corresponding agents collect, analyze, integrate or display data, to serve as feedback in adapting the process. Amadeus separates event specification from agent specification and supports both static and dynamic event interpretation, which enables users to specify agents corresponding to events and thus change the process instrumentation while the process is in progress.

Provence [140] is a process monitoring system that notifies users when there are troublesome divergences from the process model. Implemented at AT&T Bell Labs, it uses MARVEL as its process engine together with proprietary event-monitoring and file system overlay facilities, to non-intrusively track activities by users conducted through the conventional UNIX operating system. The event-monitor notifies MARVEL when any of the specified events occur on designated files or directories. Then MARVEL's original process engine attempts to execute the rule corresponding to that event. If it is not possible to satisfy the rule's condition through backward chaining, a warning is generated; notice that the event (and thus the process step) has already occurred by the time MARVEL is invoked, so enforcement is impossible.

While many PCEs support more than one process assistance mode, we know of only one other (proposed) system that could potentially support an arbitrary range of modes: The gist of ProcessWall [98] is that the process state is maintained by a central process state server with a predefined representation for steps, called *tasks*, hierarchical breakdown of tasks and control flow among tasks. The types of task input and output parameters can be specified, and bound across multiple tasks. Then multiple clients can manipulate and advance the state by enacting process fragments. *Process constructors* construct process specifications using the ProcessWall primitives, unfolding the process as it goes, and *process constrainers* enforce additional constraints on the process. The process state server is thus analogous to the blackboards of many AI systems. Special process constructors could be designed to support process planning and simulation. Similarly, process monitoring and instrumentation can be supported by relaxing and augmenting, respectively, the process constrainers.

8.9 Conclusions

We identified a second form of knowledge important to process-centered environments, beyond the process model itself, namely the process assistance policies to be adopted by a particular project or user. We introduced an approach to implementing such a knowledge-based process engine, assuming rule-based process modeling. This approach is currently being implemented in AMBER, a successor to our MARVEL system. We described our implementation of process guidance and delegation, reusing as many facilities as possible from the existing MARVEL code. Further, we explained our designs for process planning, simulation, monitoring and instrumentation, where again we were able to exploit many of MARVEL's original capabilities.

Chapter 9

External Process Server Component

Abstract

We present a model for developing rule-based process servers with extensible syntax and semantics. New process enactment directives can be added to the syntax of the process modeling language, in which the process designer may specify specialized behavior for particular tasks or task segments. The process engine is peppered with callbacks to instance-specific code in order to implement any new directives and to modify the default enactment behavior and the kind of assistance that the process-centered environment provides to process participants. We realized our model in the Amber process server, and describe how we exploited Amber's extensibility to replace Oz's native process engine with Amber and to integrate the result with a mockup of TeamWare.

9.1 Introduction

The essential concept underlying Process-Centered Environments (PCEs) is language-based extensibility. That is, each PCE provides a language in which users specify the desired tailoring of the system's behavior to their particular needs and requirements. In other words, they represent the process in the process modeling formalism provided by the PCE, and this process model is then interpreted or executed by the PCE's process enactment engine. Such "first-order" extensibility has been widely investigated for a decade or so, and several major paradigms for process modeling formalisms have been investigated, including rules, Petri nets, grammars, task graphs, and imperative code [132].

A related "second-order" kind of extensibility involves the ability to modify or re-engineer the process, perhaps dynamically while a given process instance is in progress. Termed process evolution, this concept has also been investigated in recent years in the software process community [154]. Reflection has been one influential language-based approach to evolution [10].

Such first- and second-order, fixed-language and fixed-interpreter, extensibility is inherently limited, however, in two major respects:

- *Language* — The expressive power of the particular process modeling language and its computation model determines the scope of extensibility, even when *in vivo* evolution is supported. That is, the assistance the PCE can give the user(s) in carrying out a given process is a priori restricted when the process model is written, or modified, to those processes that can be defined in the language and how that language is enacted by the process engine. For example, process-wide constraint enforcement is readily supported in most rule-based process formalisms via overloaded (data type-specific) pre-conditions on primitive operations (like OzWeb's `read` and `write` [122]), whereas constraints usually must be specified on a per task basis in Petri nets (e.g., predicates in FUNSOFT nets [90]). On the other hand, modular process hierarchy, which is built-into many (extended) Petri net formalisms, may not be directly supported in rule-based PCEs.

  While some extensions to the basic language paradigm may be done at the time the language is designed (such as extending Marvel rules with control annotations that indicate partial rule chains should be enacted as all-or-nothing transactions [14]), clearly not all desirable functionality may be envisioned *a priori*, when the language and its engine are designed and implemented. The inherent challenge of extensibility lies in the fact that the added-on feature(s) had not been thought of at the time of the initial design, or else they would already be in the language/system in the first place.

- *System* — Process modeling may be viewed as supplying a language-based application programming interface (API) to the services provided by the PCE. Again, the single and fixed API, and single and fixed interpreter or execution engine, effectively limits the capabilities of the system with regards to its behavior in supporting human process participants. For example, adding process monitoring, e.g., for measurement purposes, may require addition of interception techniques and logging facilities to track and record process activities. Adding guidance support, e.g., to notify users when tasks become enabled and allow them to select among currently enabled tasks, may demand addition of an agenda mechanism in which to store some representation of the enabled tasks [224].

  Such extensions might be written directly in an imperative process programming language like APPL/A [219], but for most declarative process modeling formalisms adding on such functionality necessarily involves modifying the underlying process enactment engine and/or integration with other (sub)systems, independent of extensions to the language per se. In other words, the process modeling API generally does not provide the primitives necessary to substantially change the mechanisms by which the PCE supports process enactment.

There are two main alternatives for advancing to "third-order" beyond the fixed single-language/single-interpreter extensibility: One is to eliminate the "singleness", through a *multi-lingual* approach, and the other is to eliminate the "fixedness", through a *meta-lingual* (henceforth metalinguistic) approach. These approaches are not mutually exclusive, they could in principle be combined, but we do not address this here.

Multi-lingual extensibility can itself be sub-divided into two different approaches: The first enables multiple languages to co-exist as peers, their independent process engines interfacing to a common process state (or enabled process task) representation, in the style of ProcessWall [98], perhaps but not necessarily with an external task scheduling mechanism [174]. The second approach is based on layers, with translators from higher to lower levels and a "process assembly language" at the bottom. We have begun investigation of the former approach, see [30], and studied the latter extensively as described in [182, 101, 127].

While enhancing the level of abstraction and potentially realizing the various enactment models intended for the different languages, the multi-lingual approach is ultimately still dependent on the capabilities of the underlying interpreter (and/or the assembly language) in the layered case, or on the process state/task server, in the peer case.

In our *metalinguistic* approach to "third-order" extensibility, the basic process modeling language can be extended with new syntax to include user-invented process enactment directives, and the interpreter can be augmented with new semantics to implement the new directives, to interpret the original syntax in a multiple different ways, and to add new process assistance services. This approach is analogous to metalinguistic abstraction in Scheme [71]. Then tailored instances of the PCE can be employed in a great variety of applications, including those not envisioned at the time the system was originally developed. It is important to note that here we generically extend the process *system* itself (i.e., the PCE), as opposed to refining, or evolving, a specific process *definition* (e.g., as in [9]).

Metalinguistic extensibility introduces difficult technical problems. In particular, the process interpreter must be designed to allow "deep" insertion of, and a structure for invoking, independently-written code that changes its internal behavior — without affecting or conflicting with built-in capabilities or other external extensions.

In the rest of this chapter, we focus on a particular process paradigm, i.e., rule-based, to which we have applied our metalinguistic approach. While we believe that metalinguistic extensibility is, in principle, generically applicable to other popular process formalisms, we have no supporting evidence yet. We demonstrate here only that our metalinguistic approach is very effective for extending the process assistance afforded by rule-based PCEs.

Section 9.2 describes the application of our metalinguistic approach to rule-based PCEs in general, and the range of potential extensibility adaptations. Section 9.3 discusses the mechanisms required to realize the metalinguistic approach, and presents an extensible rule-based process server, Amber, in which such mechanisms were implemented. Section 9.4 illustrates Amber's extensibility with respect to two different kinds of process enactment functionality: one that added multi-server connectivity and process interoperability in the style of Oz [27, 21], and another that added more intuitive process modeling (than plausible with rules) as well as on-line process visualization by integrating Amber/Oz with a mockup of the TeamWare research prototype from the University of California at Irvine [35]. Section 9.5 summarizes the contributions of this research.

9.2 Spectrum of (Rule-based) Metalinguistic Extensibility

Rules come in many forms. Some kinds have two parts (condition and action, or logical precedent and consequence), while others have three parts (condition, operator and effect, or event, action, postcondition). Some systems support only forward chaining, some only backward chaining (or inferencing), some both; AI planning systems effectively simulate chaining to draw up a specified plan. To generalize the discussion on rules (while focusing on PCE rules), however, we assume that each rule-based process modeling language (PML) has the following constructs:

- A *condition*, which specifies a predicate to be checked before the rule is executed. The condition may be formed over local or global process state variables as well as system variables.

- An *activity* that encapsulates the actual process step modeled by this rule. An activity typically involves invocation of an external "tool" application, on data which may be modeled in the PCE (in PCEs that support data integration) or also external, and by (possibly a set of) users which may or may not be designated in the body of the activity (e.g., by roles).

- A set of *effects*, which specify the assertions to be made on the process state as a result of the activity invocation. Although some rule formalisms combine activities and effects as "actions", it is usually more convenient to separate these sections in PCEs due to the fact that activities are typically external to the PCE engine.

- *Chaining* policies, which determine the control-flow of the process, i.e., when and how one rule invokes automatically other related rules as a result of a logical *matching* between them, as well as means to manually restrict automatic invocation. ¿From PCE perspective, chaining is effectively the process enactment mechanism. Matching may be intentionally formed by the process modeler who wishes to bind several activities, or automatically *inferred* by the rule interpreter, although they are in general indistinguishable from the process engine perspective.

Due to the high-level nature of rules, there is a wide range of possible adaptations that can be made with respect to process enactment. (Recall that we discuss here extensions to the language and/or interpreter which affect *all* process instances, as opposed to tailoring a specific instance.)

The first extensibility aspect concerns types of rules and their special properties. Rule type extensibility is open-ended. For example, to add guidance support, a special kind of "guidance" rules may be introduced, which entail special runtime support that may be supplied via callback functions (see Section 9.3. Another kind of rules may weaken the constraining notion of condition and continue execution even if the condition is not satisfied (e.g., to permit and monitor process exceptions). In order to support such extensibility, the original language must have a basic rule categorization mechanism into which new "basic" types can be added.

The second aspect is the chaining directions and modes. As mentioned above, the two basic directions are *backward* chaining, i.e., firing rules which may satisfy an unsatisfied condition, and *forward* chaining, i.e., firing rules whose condition became satisfied (or enabled) as a result of asserting the effects of another rule. One extension in systems with one direction might be to add the other direction. However, such modification would be in general very hard to attain as an afterthought, as it requires to essentially reimplement the core of the interpreter. On the other hand, bi-directional chaining allows for two other hybrid extensions, both of which can be extended: forward chaining during backward-chaining, and backward chaining during forward chaining. The former may be attempted when the effect of a rule satisfies the conditions of other rules. Similarly, the latter may be used when a condition is partially but not completely satisfied by the previous rule's effect. Note that the extension may or may not involve syntax changes, depending on whether the chaining modes are selectively or unconditionally applied, respectively.

A third aspect combines the two above aspects, namely, static (matching) and dynamic (chaining) relationships between rules. An obvious static extension is to adapt the valid matchings between (pre-existing or extended) types of rules. For example, an interpreter extended with guidance rules may only allow guidance rules to be associated with other guidance rules or with mandatory system-exception rules but not with other ordinary rules. Similar extensions may include syntax to turn-on and turn-off chaining, wholesale or on a per-rule basis.

Another aspect of inter-rule relationship concerns the dynamic property of their execution. For example, the capability to *atomically* execute a set of linked rules may be added. This may involve syntax notations to define and or derive the atomicity boundaries, but ultimately requires interaction with and support of a transaction manager. Besides the fact that such transaction manager component must exist (or be implemented) in order to support atomicity, the interesting point from extensibility perspective is that the interpreter must enable interface to other sub-systems in the PCE, such as the process data- and transaction- managers, as well as enable integration with external sub-systems.

The final extensible property is the order of rule evaluation. Some rule systems employ breadth-first invocation, i.e., firing all enabled rules at one iteration, followed by all subsequently enabled rules in the following iteration, and so forth; other rule systems operate in a depth-first mode, firing an enabled rule followed by all rules which have become enabled as a result of its effects, and so forth. As with chaining direction, adding from scratch a basic evaluation-order would be hard, but extending the basic algorithm, e.g., to include both modes and supply syntax to determine which mode to apply, is a feasible extension. Another ordering policy may prioritize rule execution based on (extended) rules types.

9.3 Mechanisms for Metalinguistic Extensibility in Amber

Several key design issues enable Amber to support metalinguistic extensibility, i.e., to extend the syntax and semantics of its base language. To focus our discussion, we ignore many other aspects of the language/interpreter which are not closely related to supporting extensibility (for a complete account of Amber, see [182]).

Syntactic extensions can be made (only) by means of *rule annotations.* Annotations are strings that can be attached to the different sections of the rules and affect the (default) behavior before, during, or after the execution of the rule-section as well as the default chaining behavior into or from the rule. Once they are added to the language, they become "new" keywords. For example, a `weak-enforce` annotation in the condition of the rule may only raise a warning message when a condition is not satisfied, but otherwise continue the execution of the rule. Note that by introducing such annotation we give the *option* for process modelers to use weak-enforcement, but don't change the default behavior. One can also change the default of its Amber instance to be weak-enforcing, but such a change does not require any syntax changes, only semantic ones. The restriction to use annotations as means of syntactic extensibility provides a clean and relatively simple extension-interface with the interpreter, because the basic rule structure, and thus the skeletal logic of the rule-processor, are preserved. As a result, the number of entry points for code extensions is reasonably small, simplifying the extension task while still enabling to insert arbitrarily complex functionality in each of these entry points, as will be seen in Section 9.4.

Thus, all *semantic* extensions, including but not limited to those which correspond to syntax extensions, are also constrained to be made at well-defined specific entry points in the interpreter, essentially to ease the extensibility task and avoid complex control changes which might require knowledge of the interpreter-internals. Note that if such intimate familiarity is required, such "extensibility" is mostly worthless because it is limited only to the implementors of the interpreter. Recall that our goal is to enable process *administrators* to tailor the language/interpreter to fit their special needs.

To facilitate such modular extensibility, the interpreter is *iterative*, rather than a recursive one. (This is also the reason why Amber's rule interpreter was largely rewritten although the basic language is similar to the Marvel and Oz rule languages [108]). While the latter may be viewed as more natural for implementation of rule-processing due to the recursive nature of chaining, it is far less extensible due to the deeply intertwined and inter-dependent rule phases. Instead, the rule execution algorithm iterates over a sequence of a fixed rule-phase dispatches, discussed below.

Process engine extenders can insert/revise/replace their own functionality *between* phases, using a table-driven *callback* mechanism. The callbacks are made to a *mediator* that tailors the process engine's semantics and interfaces to other environment components, in a similar fashion to [234, 169]. The callbacks can interface with other sub-systems, Callbacks can break the sequential execution, and access, in a controlled manner, internal state of the interpreter. The proper callback function to invoke is determined by a combination of the current rule-phase, the (static) rule annotation (if any), and the (dynamic) state of the rule (such as whether the rule is in backward or forward chaining). This means that callbacks reside in a multi-dimensional function array. An interesting aspect of the callback array is that even the number of dimensions may be extended to allow for multiple state attributes.

Finally, since Amber supports context-switching among multiple tasks operating concurrently or sequentially on behalf of one or more clients, a mechanism is required to be able to extend the rule execution ordering policy. This is made possible by a parameterizable *multi-priority* queue, where a rule's priority is determined by its type, as defined by the (extended) rule annotation or by the default setting. Thus, by adding a rule type annotation and assigning to it a priority, the execution ordering can be extended.

We proceed with an overview of the language followed by the full rule execution algorithm.

9.3.1 Language Overview

Amber's PML is based in part on the Marvle Strategy Language (MSL) first developed for the Marvel process-centered environment [123] and later extended in Oz (with mostly semantic changes); most of the syntax, except for the extensible annotations, was previously elaborated in [13]. The PML incorporates an object-oriented data definition sublanguage for defining classes whose members represent process state and product artifacts (design documents, source code, test cases, etc.), and a rule sublanguage that specifies the actions that can be taken by a user or by the environment.

The fixed portion of the rule syntax is similar to the generic structure mentioned earlier, consisting of a rule signature (name and typed formal parameters); a binding section for retrieving object that are related to the parameters (derived parameters); a condition, specified as a compound first-order predicate logic over (derived and regular) parameters; an activity that indicates a shell script *envelope* which interfaces the PCE to external tools and executes with specified (possibly transformed) arguments; and a set of mutually exclusive effects, each of which matches exactly one of the possible return values from the tool envelope.

An Amber process server loads a collection of rules when it starts up; different Amber instances may thus interpret different rule sets, representing different processes. The Loader utility parses a rule set and translates it into an efficient internal form, basically a rule network whose nodes are rules and whose edges represent matches between the condition (or bindings) of one rule and an effect of another; the network thus reflects all possible chaining.

The rules of a process form the command set or interface of the Amber instance. The user client or an encompassing program issues requests to Amber that result in one or more rules being instantiated and evaluated. Amber supports by default both backward and forward chaining, as well as backward during forward and forward during chaining modes, although either of these modes can be altered or removed, either optionally (by introducing rule annotation) or globally (by changing the default behavior).

Figure 9.1 shows an example rule that triggers code inspection of a C file with respect to its design document. This rule is adapted from an Oz demo environment for the ISPW9 example process [176]. The rule specifies that the review is performed whenever either of the conditions (a buggy C file has been revised or a bug has been found in a C file) are satisfied. The assertions in the effects trigger further chaining that results in a groupware document inspection application being run, assisting the designer and coder to together inspect the code.

Notice the `automation forward` annotation which is attached to each predicate in the condition and each assertion in the effect. In Amber, each chaining type (in this case `automation`) is followed by `forward` and/or `backward` to indicate whether the chaining type applies during forward vs. backward chaining, respectively (i.e., `forward` and `backward` are built-in, but `automation` is not). Multiple annotations may be attached to the same predicate or assertion, e.g., both forward chaining into and backward chaining out of the same predicate may be supported, and/or multiple chaining types may be indicated. If no annotation is given, then normally there cannot be chaining to/from that predicate or assertion. This can be changed in a given instance, to default to "on" rather than "off" for any or all of the instance-specific chaining types (in which case `no_chain` annotations would need to be introduced). Particular chaining types may require various arguments. For example, we have devised a new chaining type we call `guidance`, which works like `automation` except that after instantiating a rule and satisfying its condition, the rule instance is placed into a persistent agenda ("to do" list), so one argument is *whose* agenda (user or group) [224].

9.3.2 Amber Rule Execution Algorithm

There are two auxiliary data structures, in addition to basic rules, that are used to manage rule execution: tasks and bulges. A *task* is a set of rules, together with all of their forward and backward chaining implications. It represents the context for all the rules invoked as the result of a top-level rule selection from a client (a human user or program). A *bulge* is also a set of rules, which represent a callback-specifiable dependency among a collection of rules, where either they must be executed in sequence, or a single rule must be chosen from the bulge to run while the rest are discarded.

Amber's rule execution algorithm consists of two intertwined parts: rule selection (or scheduling) and rule execution. During the execution of a rule, it may have to be suspended when waiting. In this case, its state is preserved, it is context-switched, and a new rule, if any, is selected using the selection algorithm.

Rule Selection — New tasks (which correspond to newly issued commands) are always placed in the top-level (i.e. highest-priority) execution queue, to optimize interactive (user-issued) tasks. This policy is in fact hard-wired (perhaps unnecessarily). When a task has been instantiated, the rule interpreter runs the task until such time as it has no more rules ready to execute. Individual rules are marked as either "runnable" or "waiting". A task may not have any runnable rules because its has finished, or all of its rules are waiting, generally for either an activity or another rule to complete. Concurrent tasks are interleaved at the natural breaks afforded by activity invocations and backward or forward chaining.

If a runnable bulge is found, one of its runnable rules is selected for execution, and no more bulges are considered in this cycle. Each bulge is also checked to see if it contains a rule that is waiting for the completion of an activity. If no runnable bulge is found, but there is a waiting bulge, execution of the entire task is suspended to wait for the activity. Any rule that may remain to be executed is in a lower-priority queue than the waiting rule.

Rule Execution — As outlined earlier, rule execution consists of fixed phases, separated by callback entries.

1. *Enque* — A rule is instantiated with parameters and placed in a bulge in the appropriate queue, according to its priority.

2. *Begin* — An instantiated rule is selected from the current bulge.

3. *Bindings* — Local (to the rule) variable bindings are established via queries to the environment's data repository.

4. *Condition evaluation* — The rule's condition is evaluated. Backward chains may be instantiated and enqueued during this phase. A callback function may filter certain entries from being enqueued, or modify the entries to be enqueued.

5. *Waiting for backward chaining* — If required by the failure of the condition in this particular Amber instance, the rule interpreter attempts to perform all possible backward chains from the rule's unsatisfied predicates, until the rule's condition is satisfied or all possibilities have been exhausted. (This is the essence of `automation backward` chaining, which is built into Amber as the default chaining type for when chaining is permitted at all; a given Amber instance may completely disallow any form of chaining.) Clauses in the rule's condition are considered in the order determined by a chaining callback function, and are never reconsidered after satisfaction or exhaustion of backward chaining possibilities, to prevent infinite cycling (this decision could easily be changed, but is not currently parameterized by a callback).

6. *Activity initiation* — Once the rule's condition has been satisfied (or other implementor-defined requirements for "success" have been met), the rule's activity (a tool envelope) is sent to the originating client with the arguments derived for it in the rule.

7. *Activity completion* — The client returns a status code and other return values at an arbitrary later time when the tool envelope terminates.

8. *Effect assertion* — When the activity finishes, one effect is asserted according to the status code. etc.). Forward chains may be instantiated. (Automatic enactment of those chains is the essence of `automation forward` chaining, which is built into Amber as the default chaining type for when chaining is permitted; a given instance of Amber may support `automation forward` chaining, `automation backward` chaining, both or neither.) The ordering of forward chains emanating from a given effect is determined by a chaining callback.

9. *Waiting for forward chaining* — After an effect has been asserted, the rule is retained during any forward chains emanating from it, from its children, etc. Amber does not use the ancestors for anything, but another component of the environment might, e.g., in our Oz integration the transaction manager treats `atomicity` implications of a rule as nested subtransactions [102].

10. *End* — Clean up after chaining, e.g., free memory allocated to the rule and its chains. This is done *after* the rule *and* its forward and/or backward chaining "children" have finished, so the data structures are available for reference throughout the full chain.

Each rule instance contains a `state` field indicating one of these phases. Whenever a rule is invoked, the Amber rule interpreter runs through these states, performing each of the phases in turn, incrementing the state after each step, and invoking the phase-specific callback. Note that the waiting for backward chaining and condition evaluation phases may cycle, as each clause in the condition is considered. The callback functions may also request repetition of a phase or skipping ahead (e.g., to `end`, to terminate a rule early). Each phase callback is passed a pointer to the current rule instance, and can modify its `state` field as well as access the tree of parent and child rule instances through Amber's application programming interface (API) [143]; the rule also contains a "work area" where the callbacks can store a block of their own data for use by other callbacks.

9.3.3 Amber Callback Interface

Amber's rule execution phases are illustrated in Figure 9.2 Each phase has two callback functions: The *before* function is invoked before that phase is initiated; its return value determines whether or not the phase is actually performed. The *after* callback is executed after the phase completes, and is passed the return value from the rule interpreter's phase execution. Callback functions may skip or repeat a phase, augment a phase with additional wrap-around code, replace the phase completely with instance-specific functionality, or jump to any other phase. The reason for separating an after callback of one phase from the before callback of the next phase (as opposed to combining them) is due to the fact that callbacks can break the sequential execution and therefore a before callback may be invoked even though the previous phase (in the sequential order), and thus its after callback, have not been invoked.

We have also found it important to designate different callbacks for forward vs. backward chaining and for different chaining types, e.g., an instance implementor might not want to allow `automation forward` chaining during `automation backward` chaining, whereas he/she would almost certainly want to allow `atomicity forward` chaining during `automation backward` chaining (otherwise all atomicity guarantees would be thwarted). Thus there are separate callback functions for both forward and backward chaining with respect to each chaining type. Not all of the possible callback functions will be used in any particular Amber instance; some of them may never be used, but all options are provided in the interest of flexibility. The set of callback functions are specified in Amber's `action` array of function pointers. Currently this mediator code is linked into the Amber instance at compile-time; an improved version would support dynamic linking. Additional details can be found in [182].

9.4 Sample Extensions

9.4.1 Multi-Process Collaboration

Oz is a multi-site PCE that enables collaboration between heterogeneous and autonomous process instances running on homogeneous process engines. Each Oz environment has its own process model, data schema, objectbase and tools. User clients are always connected to their one "local" server and may also open and close connections on demand to "remote" servers. Servers communicate among themselves to establish *Treaties* — agreed-upon shared subprocesses automatically added on to each affected local process, and to coordinate *Summits* — enactment of Treaty-defined process segments that involve data and/or local clients from multiple sites. We stretch the International Alliance metaphor, since Treaties among sites precede and specify Summits rather than *vice versa.*

A Treaty consists of a set of shared rules exported by one site and imported by one or more other sites. A Summit occurs when a Treaty rule is enacted with actual parameters from two or more sites, as follows:

1. A user client selects the rule and provides arguments from its local objectbase and any of its open remote objectbases. The client's local server is said to be the *coordinating* site.

2. The rule parameters and variables are bound by the process engine at the coordinating site, with remote data automatically transferred to and cached at this site as needed.

3. The rule's condition is evaluated and any consequent chaining is performed by each allied environment with respect to its own data and according to its own local process. In particular, if a condition predicate is not currently satisfied on its argument(s), i.e., parameters and/or variables, the server where the offending object or objects reside performs backward chaining according to its own process rules to attempt to satisfy the predicate.

4. Once the condition is satisfied, the rule activity is run by the initiating user client.

5. When the activity completes, one of the rule's effects is asserted. Any implied chaining is performed at each site, again with respect to its own data and according to its own process. In particular, conditions of rules in the environment's own process model may be satisfied by the assertions, so these rules are triggered, perhaps satisfying the conditions of other local rules, and so on.

6. After all such chaining has completed and the sites have synchronized, the original Summit rule may itself forward chain to one or more Summit rules. Then the cycle repeats, enacting forward chains among Summit rules in depth-first order.

The interleaving of local and Summit forward chaining is actually more complicated than presented above. First all local `atomicity` chaining is completed, then all Summit `atomicity` chaining, followed by local `automation` chaining and finally Summit `automation` chaining. The motivation for always completing local chaining of a given chaining type spawned from a Summit rule prior to any Summit chains due to that same rule, of the same chaining type, is to guarantee local consistency prior to initiating global operations. This is analogous to two-phase commit in distributed transactions, but also applies to non-`atomicity` chaining.

Although Oz had its own process engine, we decided to replace it with Amber to enable site administrators to extend their process engines. The interesting aspect of Amber/Oz from the extensibility perspective is that Treaties and Summits were added using the extension protocol rather than modifying Amber itself. We focus here on Summits.

9.4.1.1 Amber/Oz Implementation Details

Neither `atomicity` nor `automation` chaining, nor the concept of Summits, are built into Amber in any way. `atomicity` vs. `automation` is specified in the extended syntax, i.e., predicate annotations, and the semantics are implemented in the mediator callback code. Altogether, eight chaining types were defined in Amber's `action` array, from highest to lowest priority: Local `atomicity forward` chaining; Summit `atomicity forward` chaining; Local `automation backward` chaining; Summit `automation backward` chaining; Local `automation forward` chaining; Summit `automation forward` chaining; User-invoked local rules; User-invoked Summit rules.

Several callback functions work together to realize Summits. `after_eval` is invoked when a rule condition evaluates to false. It checks whether the rule is local or Summit, i.e., the rule is defined in an active Treaty and the rule's arguments include at least one remote object. `after_eval` does nothing for local rules. For Summits, the coordinating site requests remote backward chaining at each affected site, whose own object(s) failed a condition predicate. `after_eval` set the rule frame to a waiting state until all the remote backward chains complete. It flushes remote objects used by the rule from the objectbase cache before starting up remote chaining, since the remote chaining may modify these objects. `after_eval` is one of several callback functions where the Amber/Oz mediator interfaces to the environment framework's database.

`after_bc` is called at the end of a backward chaining cycle, to attempt to satisfy a condition predicate. If there is a failure of local backward chaining, for any reason (e.g., there might be a concurrency control conflict regarding necessary arguments for one of the chained-to rules), that failure is reported to the user client. In the case of Summit rules, however, remote backward chaining is still performed regardless of the success or failure of local backward chaining at the coordinating site. An alternative model would attempt any local backward chaining needed at the coordinating site first, and only if it succeeded in satisfying the relevant predicates, would any remote backward chaining (to fulfill the remaining unsatisfied predicates) be attempted. But that would reduce parallelism, and not necessarily reduce the apparently "unnecessary" work incurred via the remote backward chaining in such cases. We presume that in most cases the user really does intend to perform the requested Summit rule eventually, and will make arrangements for eventually fulfilling all the prerequisites and try again.

`set_chaining_types` is called when instantiating forward and backward chaining, with the list of new rule frames generated by chaining from a particular rule. When that chaining was ultimately triggered by a Summit rule (that is, a rule running as part of a Summit on a non-coordinating site), it sets the chaining type of the new rule frames to the Summit chaining type corresponding to the type that has been placed in the rule frames by the Amber process engine. For example, if a Summit `automation` rule frame generates `automation` and `atomicity` children, those children will have their types set to Summit `automation` and Summit `atomicity`. If it were not for this rather subtle callback function, remote rule chains triggered by a Summit at a non-coordinating site would not be recognized as part of a Summit by their local process engine.

The rest of the callback functions are concerned with transaction management. When `atomicity forward` chaining is instantiated between the currently running rule and the other rules whose conditions it satisfies, `enqueue_atomicity` is called for each chained-to rule to initialize a transaction nested within the parent rule's transaction, with the standard commit and abort dependencies between the transactions. In contrast, when `automation forward` chaining, `automation backward` chaining, or a top-level rule is instantiated, `enqueue_automation` is called. In the case where there is no parent rule frame (i.e., a top-level rule), it starts up a new top-level transaction. Otherwise, it starts up a transaction nested within the parent rule's transaction, but with no dependencies between the transactions. That is, each rule chained to during `automation` is treated as an independent transaction, that can commit or abort separately from the rest of the enclosing rule chain, rather than as a nested subtransaction where all or none of them commit.

When a rule is selected for enactment, `summit_tx` is called for Summit rules and `tx_init` for all other rules. The process engine *is* able to distinguish Summit vs. local here, due to the "Summit" keyword in the chaining annotation. `tx_init` links the rule frame and the current transaction (recall that callback functions are passed a rule frame pointer). `summit_tx` prepares for a Summit by retrieving copies of any needed remote objects not already cached, and links the Summit rule and the current transaction. (`summit_tx` is called as a subroutine by `tx_init` for a top-level Summit rule, because then the process engine cannot distinguish local vs. Summit cases.)

`rp_acquire_locks` is called after the binding phase for both local and Summit rules. It traverses the lists of bound objects in the rule's parameters and local variables, acquiring locks on all of the objects that the rule might access. `finish_rule` is called after effect assertion. In the case of a local rule, it checks whether any "child" rule frames were generated by forward `atomicity` chaining. If there are no such children, it commits the transaction associated with the rule. But if there are any atomicity children, the transaction is left open until the `end` phase callback (`commit_rule` below). In the case of a Summit rule, the subroutine starts up remote forward chaining at all relevant sites, and then proceeds as for a local rule. It sets a site counter to the number of remote sites involved in the Summit, for later reference by `commit_rule`.

`commit_rule` is called at the end of rule execution. It commits any remaining transactions associated with the rule (if it had `atomicity` chaining children when `finish_rule` was called). `commit_rule` checks if the just-completed rule was the last remaining `atomicity` child of a parent rule and, if so, it commits the parent's transaction. For rules created by remote chaining during a Summit, it notifies the coordinating site that chaining is complete. For Summit rules (i.e., at the coordinating site), it decrements the site counter used for termination detection.

Oz's process engine was directly cognizant of locks and transactions. Although Amber/Oz uses the same transaction management component as the original Oz, described in [105], Amber itself knows nothing at all about locks or transactions; all code concerned with concurrency control and failure recovery of rules and rule chains is located in callback functions (or mediator utilities called by those functions).

9.4.2 Integration with micro-TeamWare

TeamWare [235, 236] represents the process model as a task graph, where nodes in the graph define process steps or tasks. After the process engine is informed that a task has been completed, a user selects among the connected nodes by traversing an outgoing edge (there may be several alternatives, iterations, etc.). Each node is associated with a script that may launch some external tool. The work of multiple users may be guided by the same task graph. TeamWare thus provides nice process visualization features. It is well-suited to modeling process topology, the "big picture" of concern to managers and other non-technical users, as well as clearly representing the workflow path and the choices for what to do next.

In contrast, Amber/Oz's process animation (inherited from Oz) graphically shows chaining from rule to rule as the chain unfolds, and the full rule network showing all possible chaining options can be displayed. But there is no "roadmap", the user has to *know* what to do next. (Only simple single-user processes can easily be written as one long forward chain, where the user is *told* what to do next, although we describe two different experimental extensions that would support this for multiple users in [127, 29].) Amber/Oz, and the original Oz, provide powerful execution facilities (e.g., both goal-driven and event-driven automation), and sophisticated multi-user/context-switching support (with collaborative concurrency control policies achieved through the mediator interface to an external transaction manager [102, 105]). But the user interface limitation has been a severe block to serious use of Oz (and Marvel before it) by anyone other than the development group.

Thus it seems valuable to *integrate* TeamWare and Oz, to exploit the advantages of both. We did not integrate Amber/Oz with the real TeamWare, which was temporarily out of order, but instead with our toy version called micro-Teamware (μTeamWare). We plan to later integrate Amber/Oz with Endeavors [35], the successor to TeamWare, following the same integration architecture.

9.4.2.1 micro-TeamWare Integration Approach

The gist of the integration, from the process perspective, is to use μTeamWare task graphs to model the process topology, the sequencing of tasks, that Amber/Oz users otherwise have to *know* off the top of their head. The users can then look at the status of the task graph to determine what to tell Amber/Oz to do next. μTeamWare activities correspond to entry points into Amber multi-rule chains, from which backward and/or forward chaining could occur, and perhaps also any follow-on rules directly selected by the user to complete the required work. Thus μTeamWare activities would be relatively coarse-grained compared to Amber's individual rules.

We took advantage of Amber's extensibility to add *agendas* ("to do" lists), such that an enabled task intended to be performed by a particular user automatically appears in that user's agenda. The user can select an agenda entry for enactment at his/her convenience. Agendas are persistent, so that work could easily be assigned to a user who is not currently available. Finally, group (or role) agendas allow for any member of the group to select the entry, so it is not necessary to pick a particular user *a priori* when any user in the group (or who fulfills the role) will do. We had previously developed a similar notion of agendas [224], but it had never been integrated in the mainstream version of Oz — although we were able to port some of the old code to Amber. The agenda implementation, e.g., persistent maintenance of agendas, is implemented entirely in "mediator" code, not hard-wired into Amber.

The μTeamWare to Amber interface was simple: Amber's TCP/IP message interface, in the Oz mediator code, was extended so (1) a client can request that a particular instantiated rule (with parameters) be added to a specified agenda, and (2) a client can request the display of a particular agenda. Then the μTeamWare scripts, triggered when a task node is enacted, simply send messages to add rules to agendas. Regarding Amber's rule phases, the callback after the *Enque* phase jumps to the *End* phase. That is, the rule is not executed until later selected by a user from an agenda, when it goes through the phases normally.

An Oz graphical user interface client (based on Motif) was modified to display agendas in the form of menus, allowing the user to select instantiated rules from an agenda for enactment at his/her convenience. The user can request his/her own agenda, any other user's agenda, and/or one or more group agendas. The user interface also permits manual construction of agenda entries. There is currently no access control to limit agenda display or even such assignments to, say, the requisite user and his/her manager, but on the other hand there is nothing forcing a user to ever invoke any of his/her assigned tasks or preventing him/her from manually deleting them.

The Amber to μTeamWare interface was a bit more complicated: Amber needed to have some way to indicate to μTeamWare when a task node was completed and thus an outgoing edge could be followed. The obvious way to achieve this would seem to be through mediator callbacks. One model would be for the mediator to maintain state information regarding whether or not a given rule chain was triggered by selection of an agenda item that had been originally placed in the agenda by μTeamWare (as opposed to by a user manually). Then the *End* callback for the top-level rule, invoked after all chaining emanating from that rule has completed, would send a message back to μTeamWare . But this was problematic: Amber "knows" when a multi-rule chain completes, but does not "know" when a set of related chains is done. It is desirable to permit a μTeamWare task node to model larger process segments than might be appropriately implemented by a single rule chain, e.g., when edit-compile-debug iteration is needed, or for upstream activities like design, which may not map nicely to a sequence of tool invocations.

A better model, also using mediator callbacks, would be to invent a new annotation, that when included in the asserted effect, would be interpreted by the *Effect* callback function as signaling the end of the μTeamWare task. The rule containing this annotation might be automatically chained to, in cases where it is appropriate to implicitly end the task, or might be explicitly selected by a user when the task should only be ended at a human's discretion. However, this had the disadvantage of including specific knowledge about μTeamWare in the mediator code, whereas it would be preferable to keep agendas completely general — there are certainly other useful purposes for them unrelated to the μTeamWare integration.

So, we decided on a variant of the second model: instead of placing an annotation in the effect of a rule whose enactment is intended to signal the end of the encompassing μTeamWare task, that rule's activity indicates an envelope whose invocation would send the proper message to μTeamWare (such messages can also be tacked onto the end of regular tool invocation envelopes). The approach restricted all knowledge by Amber of μTeamWare and *vice versa* to their envelopes and scripts.

9.5 Contributions

We sketched an approach to extensible process enactment systems to enable addition/modification of the assistance a process-centered environment provides its users. We concentrated on the rule-based paradigm both because our experience lies there and because we had already shown that other higher-level process formalisms could easily be translated into rules for enactment. However, we plan future work to investigate extensibility for other process paradigms, notably task graphs.

We presented our realization of an extensible rule-based process modeling language and process interpreter in Amber. Amber's syntax follows closely the notation we had already developed for Marvel and Oz, with moderate changes. However, the rule interpreter was largely rewritten in a completely different style, basically iterative over a sequence of rule-phase dispatches, rather than recursive with the rule phases deeply intertwined. It was simple to insert the table-driven callbacks, and it should be relatively easy to make other parts of the interpreter parameterizable.

We had planned all along to replace Oz's process engine with Amber, and then to add various new process assistance functionality to Amber/Oz such as guidance chaining and parallelized automation chaining. But the TeamWare integration idea came much later, and demonstrates the versatility of our extensibility approach to change the process enactment model in a way we had not originally envisioned: integration with a second process engine. We cannot present it here due to space constraints, but Amber's extensibility was also a significant factor in our experimental replacement of ProcessWEAVER's native Petri net-based process engine. The power and flexibility of the callbacks mediating between the two systems was particularly critical since ProcessWEAVER is a commercial product and we had no access to its source code or internal documentation; see [183].

```
setup_review[?p:PROJECT,
     ?design_doc:DESIGN_DOCUMENT]:

 # this rule prepares for a review of
 # one of the project's C files
 # against its design document.  It is
 # triggered first when a defect
 # is found in the C file, and then
 # afterward whenever the C file
 # is revised.

(and (bind MODULE ?m suchthat
       (member [?p.srcs ?m]))
     (bind CFILE ?c suchthat
       (and (member [?m.cfiles ?c])
            (?c.bug_status = Defected)))
     (bind DOCFILE ?d suchthat
       (member [?design_doc.docfiles ?d])))
    :
(forall ?c
    (or (?c.review_status = Revised)
            automation forward
        (?c.bug_status = Defected)
            automation forward))

{ REVIEW_TOOLS init_review
    ?c.change_request
    ?c.bug_report ?d.change_request
    ?d.bug_report }

(and
    (?d.review_status = ReviewRequested)
        automation forward
    (?c.review_status = ReviewRequested)
        automation forward);
```

Figure 9.1: Example Amber Rule

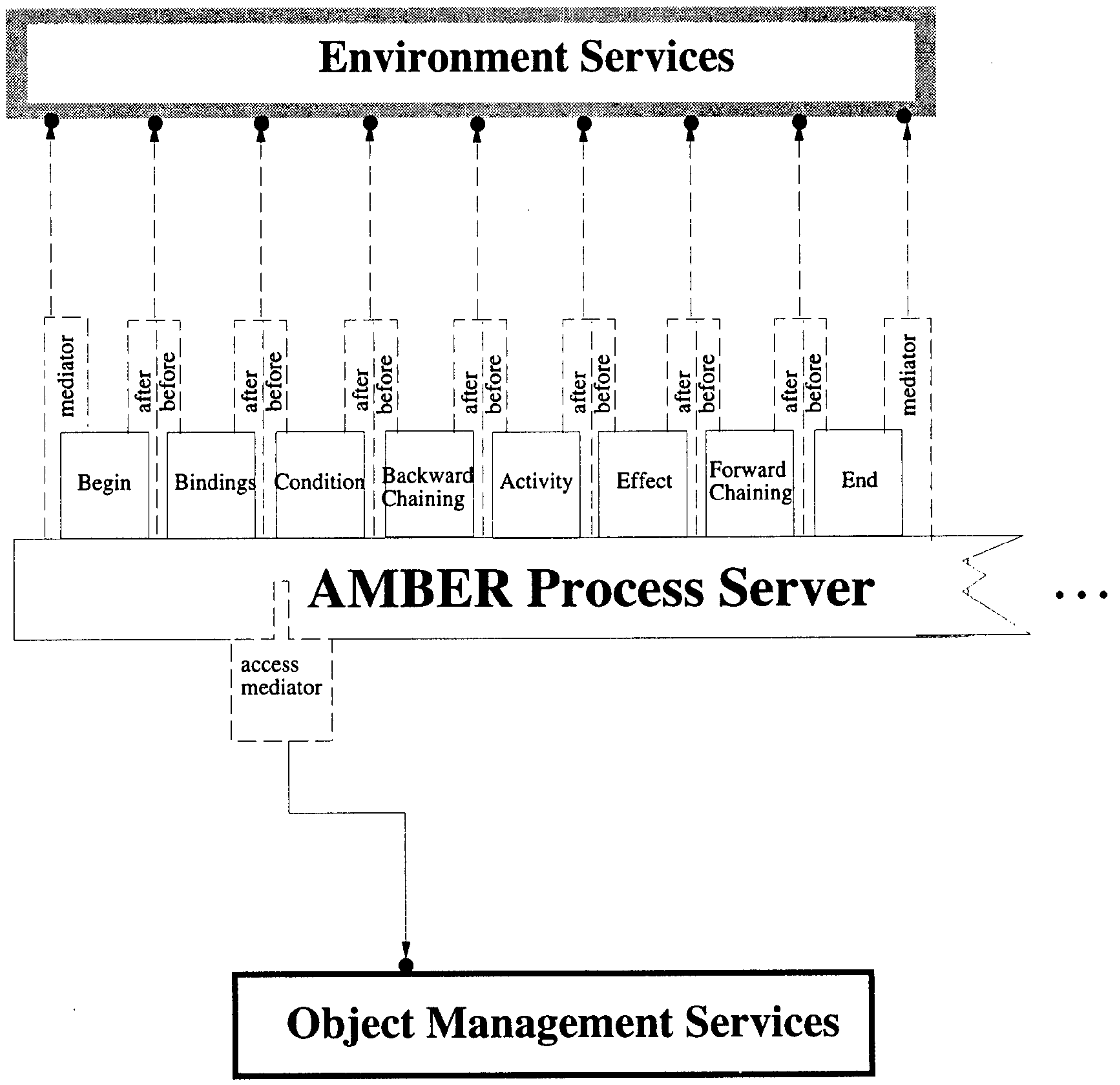


Figure 9.2: Amber Internal Architecture

Chapter 10

External Transaction Manager Component

Abstract

Layered and componentized systems promise substantial benefits from dividing responsibilities, but it is still unresolved how to construct a system from *pre-existing, independently developed* pieces. Good solutions to this problem, in general or for specific classes of components, should reduce duplicate implementation efforts and promote reuse of large scale subsystems. We tackle the domain of software development environments and present an architecture for retrofitting external concurrency control components onto existing environment frameworks. We describe a sample ECC component, explain how we added concurrency control to a commercial product that had none, and briefly sketch how we replaced the concurrency control mechanism of a research system.

10.1 Introduction

Multi-user software development environment frameworks (SDEs) need a concurrency control mechanism to detect and resolve conflicts since more than one user task may attempt to access the same data in incompatible ways. Many SDEs provide some variant of the file checkout paradigm, typically coupled with versioning, while others incorporate a database system with conventional transactions. SDE applications sometimes require "cooperative transactions" (see [16] for a survey of collaborative concurrency control mechanisms). Cooperative transactions make it possible to guarantee atomicity (rollback of an entire atomic unit if it cannot be completed) and possible to enforce serializability (the appearance that only one user task is accessing the data at a time). In addition, cooperative transactions can exploit application-specific semantics to resolve concurrency conflicts, often to enhance concurrency and enable collaborative work.

Some SDEs, however, do not provide any concurrency control at all, either because the system was initially envisioned as supporting only one-user/one-task (at a time) and later extended to multiple concurrent tasks with manual synchronization (e.g., passing the "floor" in multi-user editors [51]), or because the designers assumed that some external facility would provide concurrency control (as for ProcessWEAVER [67]). In this chapter we present an external concurrency control (ECC) architecture with our main example drawn from the latter category, where concurrency control was left for another component.

Componentized systems offer potential benefits by dividing the technical and economic responsibilities for providing services; in the case of SDEs, the ECMA/NIST Reference Model [168] specifies user interface, task management, object management, and communication components, as well as individual tools. It is still unclear, however, how one can and should integrate SDE components together to produce a coherent, usable system. We do not attempt to address the entirety of this very large problem: we are concerned primarily with architectures for integrating the task management services (TMS) component of an SDE with ECC; we also discuss integration of ECC with object management services.

In general, TMS components, as described in the literature, do not specify any particular model of what they require for concurrency control, nor do the (known) implementations provide any predefined interface to an ECC utility. Thus we must supply an interface to the ECC from TMS (i.e., entry points) that would cover the plausible range of ECC functionality, along with an interface from the ECC to TMS (i.e., callbacks) that provides a means for ECC to obtain the semantic information required to support that functionality. In particular, we took the abstract concept of mediators and made them concrete. We devised a generic ECC component with specific entry and callback points that could be directly invoked, or responded to, by a new system constructed around the component. More significantly, they are designed to be exploited by a mediator (or set of mediators) between ECC and a pre-existing TMS (above) and between ECC and a pre-existing OMS (below).

From the viewpoint of software architecture, this chapter explores the integration of independently developed, pre-existing components, in contrast to work on (1) construction of systems based on one (or a small number of) pre-existing components, with the rest of the system implemented mostly from scratch to take advantage of these components (the approach taken for PCTE [221] and many systems in other domains, such as Mach [192], Camelot [61], and database applications); or (2) building-block kits, where sets of components that can be mixed and matched are designed and implemented together (e.g., Genesis and Avoca [19]). The most significant difference compared to the above technologies is that we deal with components *whose interfaces do not match* and *whose interfaces cannot be modified to match* – much like trying to fit a square peg in a round hole. Under such circumstances, we argue, integration is possible by external mediators that overcome the mismatch between components (see [218] for a related approach that assumes components with extension languages).

We first present the requirements we have identified, both for the internal concurrency control model of TMS (perhaps brought out through a mediator) and for interfacing to an ECC component. Then we sketch our prototype ECC component, PERN [100]. We demonstrate how we applied our approach to a TMS component without its own concurrency control mechanism. This is followed

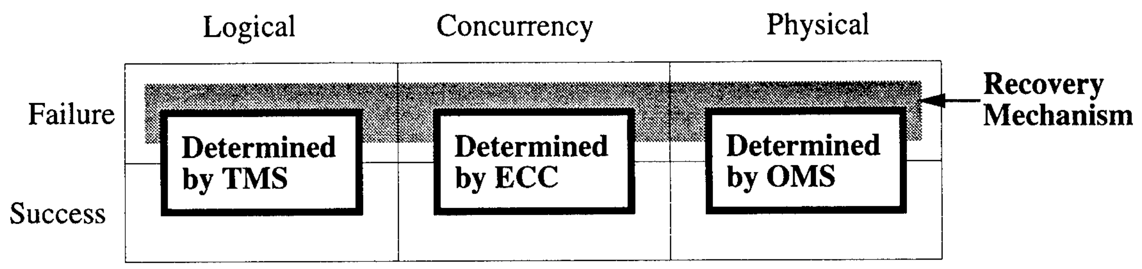


Figure 10.1: Division between logic and concurrency

by a sketch of a later study where we replaced a system's existing concurrency control mechanism, and a discussion of lessons learned. We conclude by summarizing our contributions and directions for future work. Since our focus in this chapter is on *interfacing* TMS and ECC mechanisms, we do not discuss the cooperative transaction *functionality* supported by our component (for details, see [100]); for simplicity, we assume conventional atomic and serializable transactions.

10.2 Requirements

The basic requirements imposed on TMS by concurrency control mechanisms, whether internal or external, include the following:

- *Logical units* that could be mapped to transactions. Ideally, multiple granularities would be represented: at least individual activities and full sessions as well as tasks, and possibly intermediate granules such as hierarchical tasks. This would permit different concurrency control properties to be ascribed to different levels, such as isolation during activities (or individual data accesses within activities) while still allowing collaboration within a circumscribed group during tasks (that is, group members' tasks might interleave at activity boundaries).
- *Recovery* of logical units when failures occur, typically via rollback (undo to a point before the logical unit began) or compensation (restoration to a semantically consistency state, not necessarily the same state the system was in before the logical unit began). As illustrated in Figure 10.1, TMS is responsible for determining when tasks logically succeed (e.g., goals are achieved) or fail (e.g., logical prerequisites cannot be satisfied or implications cannot be fulfilled). ECC detects when a transaction fails (e.g., unresolvable conflicting accesses among transactions so that one or more must be disallowed) or succeeds (e.g., the absence of such conflicts). Object management services (OMS) is responsible for detecting any failure to store data. Ideally, the mapping from transactions to tasks would permit the same recovery mechanism to be used for all three components.

When the concurrency control mechanism is external to TMS, there are several additional requirements:

- The ECC component must not require source code modifications or recompilation of the TMS component (this is impractical for most vendor offerings, typically provided as binary executables or run-time libraries). Ideally, the TMS component should not require code changes to ECC, although an application programming interface (API) or other parameterization mechanism should be available.
- The special-purpose mediator code, or "glue", should be relatively small, compared to the code size of either TMS or ECC. A competent system builder should be able to construct the mediators from the interface specification of TMS and ECC; that is, no special knowledge of the interior workings of TMS or ECC should be required.

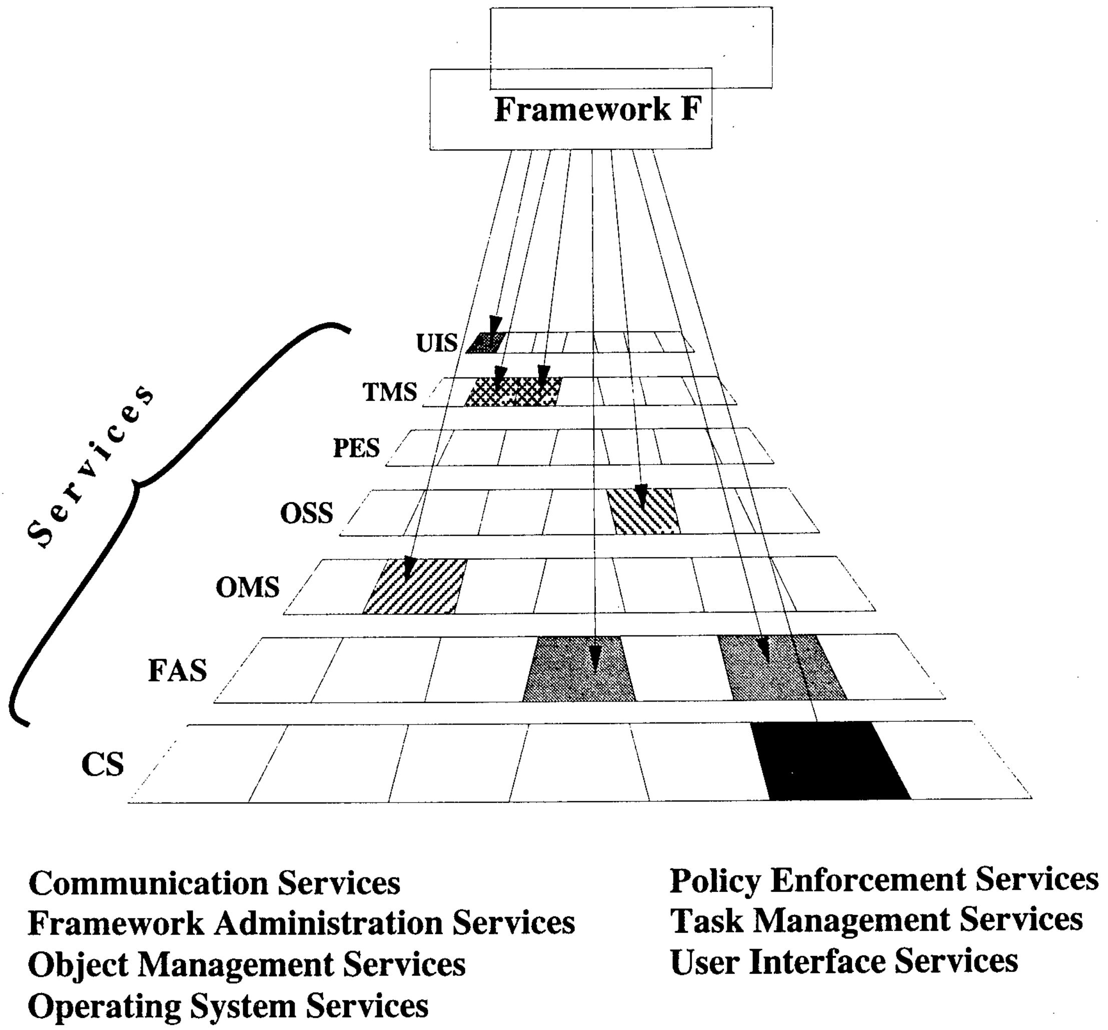

Framework F
Services
UIS
TMS
PES
OSS
OMS
FAS
CS
Communication Services
Framework Administration Services
Object Management Services
Operating System Services
Policy Enforcement Services
Task Management Services
User Interface Services

Figure 10.2: ECMA/NIST Reference Model [168]

- The performance level should be appropriate for the interfacing style of TMS. In particular, if TMS provides a library for direct linking, higher performance would be expected than one where interaction with ECC occurs over a message bus. This implies that a practical ECC should provide multiple interconnection vehicles.
- The choice among ECCs (assuming a range is available), or the decision to employ an ECC at all, should not unduly restrict the selection of other system components, most notably the object management system. There may need to be some means, however, for ECC to interact with the other components, such as through additional mediators or an extended interface provided by TMS.

10.3 ECC Architecture

To fix the terminology for this chapter, we turn to the ECMA/NIST Reference Model [168]. This model targets particular services of an SDE and groups them by functionality. Figure 10.2 shows the seven major service groupings and how a framework F provides a particular set of services from these groups. Object management services store data in any combination of file system and

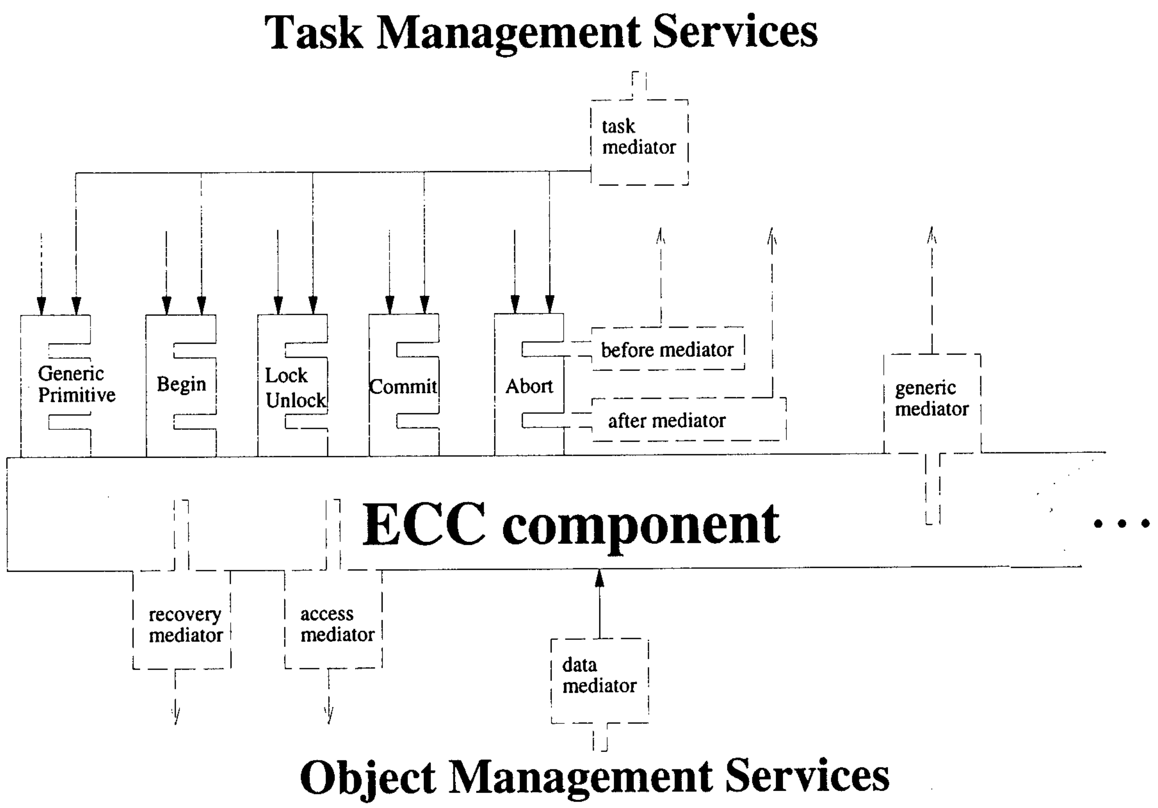


Figure 10.3: ECC component architecture

database(s) and manage access to the data, possibly using a transaction service. Task management services[1] (TMS) insulate the users from the actual details of the tools operating on the data. Generally, a TMS layer organizes user activities into goal-oriented tasks and provides a high-level abstraction for managing tasks. Unfortunately, some existing SDEs provide no particular notion of tasks except (degeneratively) as individual tool invocations. Finally, user interface services provide the appropriate abstractions for users.

SDE architectures do not necessarily match this reference model, since it is not a blueprint or a design specification. Individual SDEs may or may not be constructed from modules or components that map to the various Reference Model services. From the viewpoint of concurrency control, there are two possibilities: either a special concurrency control component is built in (i.e., internal) for each Software Development Environment as part of its OMS, or the community provides a variety of ECC components to choose from and system-specific mediators are constructed to compensate for any interface mismatch with regard to the selected ECC. In theory, the latter should involve less total work, at least if the mediators tend to be small compared to TMS and ECC. In addition, an ECC component would be more robust after being employed across a range of SDEs.

Figure 10.3 presents an abstract representation of the ECC component interface as it fits into a larger architecture. Through ECC's interface, TMS (including, for the purposes of this discussion, any ECC/TMS mediator) can *Begin* transactions, *Lock/Unlock* data items, and *Commit* or *Abort* transactions. Since object management services are external to ECC there should be no restriction on the information stored. However, ECC does assume that each data item can be uniquely referenced. This assumption is reasonable considering the increasing popularity of object-oriented databases; it also holds true for relational databases with unique key fields, and file systems with unique file

[1]The newest version of the Reference Model uses the term process management, whereas previously this was called task management.

pathnames.

ECC manages all transactions created by TMS and allows TMS (or TMS/ECC mediators) to define the composition and dependencies of transactions. Each transaction is either a "top level" transaction or has at least one parent. Through composition, TMS can create nested transaction hierarchies of arbitrary depth and breadth. Transactions may have dependencies on other transactions, such as *commit* and *abort* dependencies [42]. If transaction T_1 has a commit dependency upon T_2, then T_1 can't commit until T_2 does. If T_1 has an abort dependency upon T_2, then T_1 must abort if T_2 aborts. TMS use dependencies to group individual transactions into units.

ECC defines its interface using *mediators*, fragments of special code needed to integrate ECC with a particular TMS. Each ECC primitive has two mediators associated with it, namely *before* and *after*. When the primitive is requested by TMS, these special code fragments are executed before (and after) ECC services the request. Mediators can override a primitive request, in effect denying the operation to take place. Other mediators, such as the `access mediator`, allow ECC to interact more closely with object management services. For example, if ECC is requested to acquire a lock on a sub-item of a larger set of data items, the mediator can request intention locks on all ancestors (as in Orion [80]). During recovery, the `recovery mediator` communicates with object management services to restore the data appropriately, thus abstracting ECC from any one particular data representation.

The architecture in Figure 10.3 is open-ended and flexible, allowing for several different types of mediators. The closer the ECC and TMS interfaces match, the less need there is for mediators; an exact match would require none. New primitives can be added for a specific TMS, each with appropriate mediators; this may require modifying the ECC or switching to an existing, more sophisticated ECC. `task mediators` can work closely with TMS, translating actions at the task level into the required ECC operations; this is most appropriate for retrofitting transactional units onto a TMS that has no notion of transactions.

An example ECC component

We created the PERN component as part of a general strategy to componentize the existing MARVEL system [31]. The transaction manager from MARVEL was isolated and reengineered as a separate component. First, references to MARVEL 's TMS were eliminated from within the transaction manager, so PERN no longer made assumptions about the structure of user tasks. PERN was no longer consulted during the instantiation of tasks or their scheduling and execution. This enables PERN to manage transactions for a wide range of SDEs.

Second, external lock tables and a generic recovery mechanism were implemented in PERN , severing ties with MARVEL 's OMS. Separating the data items from the locks being held was a big step towards allowing PERN to be used with arbitrary data repositories. In addition, the generic recovery mechanism removes assumptions about the operations that can be performed within a transaction.

Finally, the mediator architecture was established, allowing interface code to be written separately from PERN . These mediators, drawn in dashed boxes in Figure 10.3, make the necessary connections to integrate PERN with TMS and OMS. The two architectures in Figures 10.4 and 10.6 show how we integrated PERN with OZ and ProcessWEAVER. It is important to note that the basic architectures are the same: in both cases mediators were written to attach PERN to the specific SDEs.

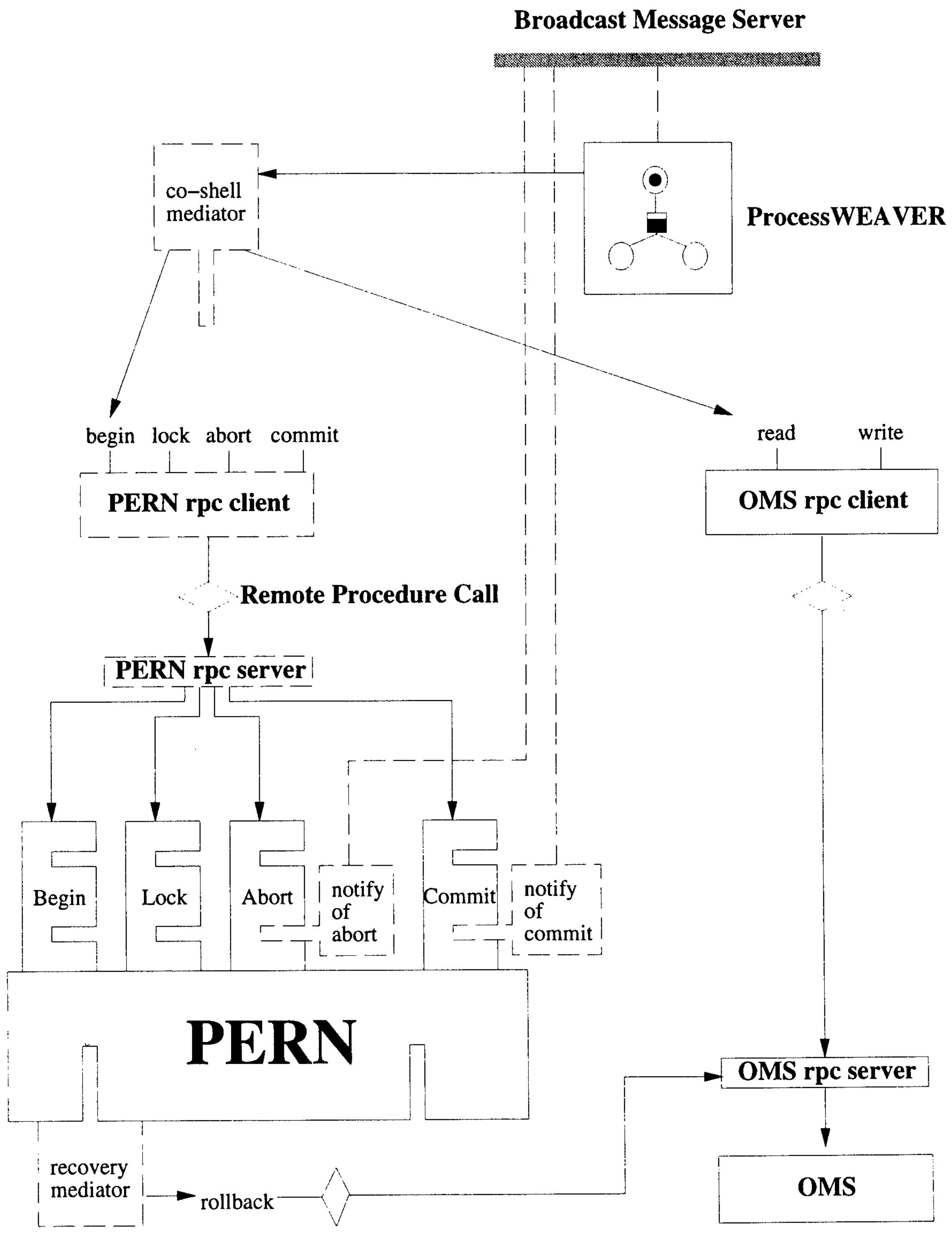


Figure 10.4: Interfacing of PERN with ProcessWEAVER

10.4 ProcessWEAVER

ProcessWEAVER [67] is a set of utilities that adds process support capability to Unix-based toolkits, including definition and execution of process models [83, 113]. ProcessWEAVER executables communicate with each other through a Broadcast Message Server (BMS).

A process model in ProcessWEAVER has two levels, the topmost being an *activity* hierarchy that recursively refines activities into sub-activities. The second level contains a set of *cooperative procedures* (CPs) each of which implements an activity or sub-activity. A CP is a transition net (a form of Petri net) consisting of places, tokens, and transitions; its state is determined by the assignment of tokens to places. Each transition has a set of input and output places associated with it. When all the input places for a transition contain tokens, the transition fires and the tokens are moved to the output places. Each transition can have *co-shell* (ProcessWEAVER's shell-like language) code that is executed (as a subroutine) when the transition fires. ProcessWEAVER doesn't have an object management service; the desired repositories, file system or database, are accessed through co-shell code.

Petri nets are excellent for explicitly modeling the synchronization of concurrent activities of cooperating agents, but there is no underlying mechanism for treating conflicting actions of concurrent, independent agents that incidentally access the same data. This distinction between concurrency through synchronization and concurrency control reveals a need for transactions in ProcessWEAVER. Since ProcessWEAVER is a commercial product without source code availability, it could not be modified – so we provided transaction support by integrating PERN , externally.

We implemented a 200 line co-shell library (see [104] for details) that defined a mediator for ProcessWEAVER to communicate with PERN . ProcessWEAVER was thus parameterized so that only individual CPs had knowledge of transactions, not the rest of the process modeling facilities and execution framework. This mediator is typical of what would be needed to integrate PERN into an SDE that provides an extension language, in this case co-shell, that makes it possible to directly augment its task management services. In our demonstration, ProcessWEAVER provides the TMS component and PERN becomes a physically separate utility working in conjunction with the other ProcessWEAVER utilities.

In [104] we discussed several alternatives for adding transactions to ProcessWEAVER, ultimately deciding upon *augmenting* a cooperative procedure with additional places and transitions. The CP in Figure 10.5, for example, is the result after manually augmenting the original transition net contained within the dotted lines; this requires only three additional places and six transitions. We envision that a preprocessor could be built to generate these augmented CPs with some user guidance, for example, in determining the appropriate starting and ending places for a CP that determine the logical unit for which the transaction is responsible.

This particular CP retrieves the balance for **account_1** and prompts the user for an amount to deposit into the account – a conventional transaction processing application. We now step through the execution of the augmented CP. A transaction is started at transition **_BEGIN_TX** through its co-shell action[2]:

```
{ Begin Transaction }
$tid = Begin (NOCOMMIT, NOABORT, TOP, ROLLBACK);
```

This issues a remote procedure call to PERN that creates a top-level transaction that can be rolled-back and has no commit or abort dependencies on any other transaction. This transition activates the original starting place for the CP, **Begin**, and also moves into the **_Idle** place. Note that the original CP (within the dashed lines) continues its normal actions. The **_Idle** place monitors the newly created transaction and has three separate transitions that are activated if the transaction commits (**_c**), aborts (**_a**) or suspends (**_s**). The condition for **_a**, for example, is satisfied if a message of type "`pern_abort $tid`" appears on the BMS. If transaction `$tid` ever aborts, the CP moves

[2]Co-shell variables start with a `$` character. Arbitrary string values, such as `TOP`, don't need to be quoted. The `Begin()`, `Access()`, and `Read_attribute()` functions are described in [104].

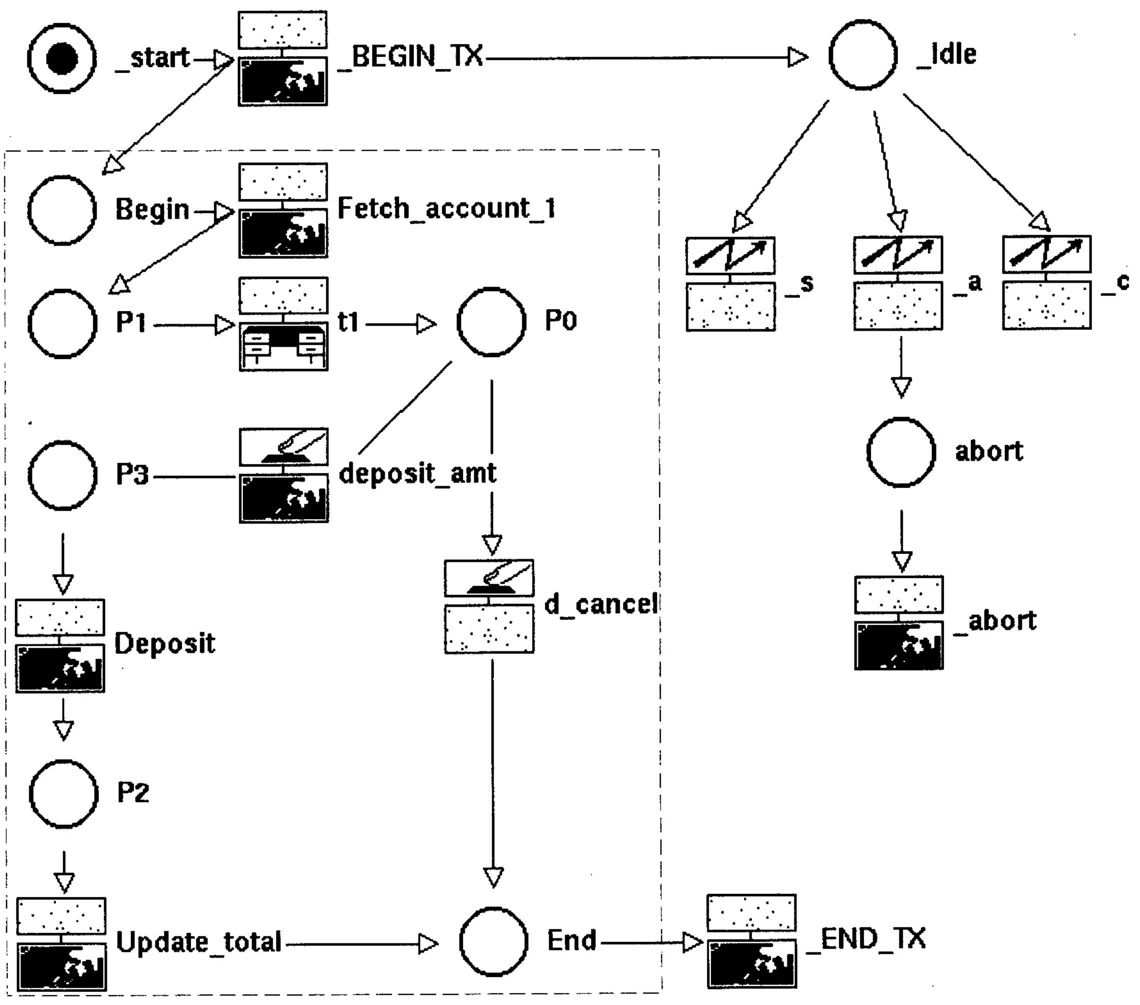


Figure 10.5: Augmented Deposit CP

into the **abort** place, which exists to allow other CPs to take action in response to this transaction abort.

For this demonstration, we constructed a miniature OMS that allowed a CP to read and write the attributes of an object. Transition **Fetch_account_1**, for example, executes the following co-shell code:

```
{ Access account-1 (through an object identifier $account_1).}
{ Request access from PERN, then get data from the OMS.}
$action = Access ($tid, $account_1, X);
$v1 = Read_Attribute($account_1, balance);
```

The `Access` function issues a remote procedure call to PERN , attempting to set a lock on the account object in e**X**clusive mode. If this request succeeds, `Read_attribute` issues a remote procedure call to the OMS to get the balance for `$account_1`.

At transition **deposit_amt**, the user determines the deposit amount to be added to the balance `$v1` (in **Deposit**) and in transition **Update_total**, the new balance is written back to the OMS and the CP moves into the final **End** place. At this point, the new transition **_END_TX** commits the transaction by issuing a remote procedure call to PERN . The *after* mediator for `Commit` sends a message to the BMS of type "`pern_commit $tid`". The transition **_c** will retrieve this message, clearing out the token at **_Idle**, and the CP will complete successfully. Note that if the `Access` function had failed, the *after* mediator for `Abort` would have sent out a `pern_abort` message, thus moving the CP into the **abort** place.

10.5 Oz

Oz [27] is a multi-site, *decentralized* process centered environment. The Oz process server is the TMS component of the Oz architecture and defines a three-level hierarchy of nested contexts. The lowest level, the *activity* level, is where Oz interfaces to actual tools (through envelopes [86]). The *process-step* level encapsulates activities with prerequisites and immediate consequences (if any) of tool invocations as determined by a process. The *task* level is a set of logically related process-steps with the combined set of their prerequisites and consequences. An Oz environment is a collection of multiple Oz *sites*, each containing its own process server that executes the local process at that site. Each site communicates with other sites through an interprocess communication layer (IPC).

The decentralized process modeling aspects of Oz allow *Treaties* to be formed between sites to define the specific collaboration that may occur among those sites. A Treaty between sites SiteA, SiteB, and SiteC, for example, defines a common sub-process, one or more process-steps, that becomes part of each site's local process and represents those fragments that involve all three local processes. The execution of such a shared sub-process by multiple sites is called the *Summit execution protocol* or Summit, for short. This international alliance metaphor emphasizes site autonomy, using Treaties to relax autonomy to the degree (and only the degree) that each local site wishes to collaborate with other sites.

Multiple users can execute tasks in each of the local process servers, independent of any simultaneously executing Summits. To prevent an environment from being left in an "inconsistent state" when a concurrency failure interrupts a Summit, the concurrency control mechanism restores the environment to a "consistent state". Oz defines consistency by grouping individual process-steps together into atomic units that must be executed in their entirety or not at all. Since each process-step corresponds to a transaction, Oz creates these units by setting dependencies between transactions.

The MARVEL system, from which Oz evolved, already provided support for guaranteeing consistency within a site, but MARVEL supported only single-site environments and had no notion of Summits. We considered two possible approaches to providing transaction semantics for Summits. The first required building a special purpose transaction manager for Oz . The better alternative – and the one implemented – attached a copy of the PERN component to each Oz process

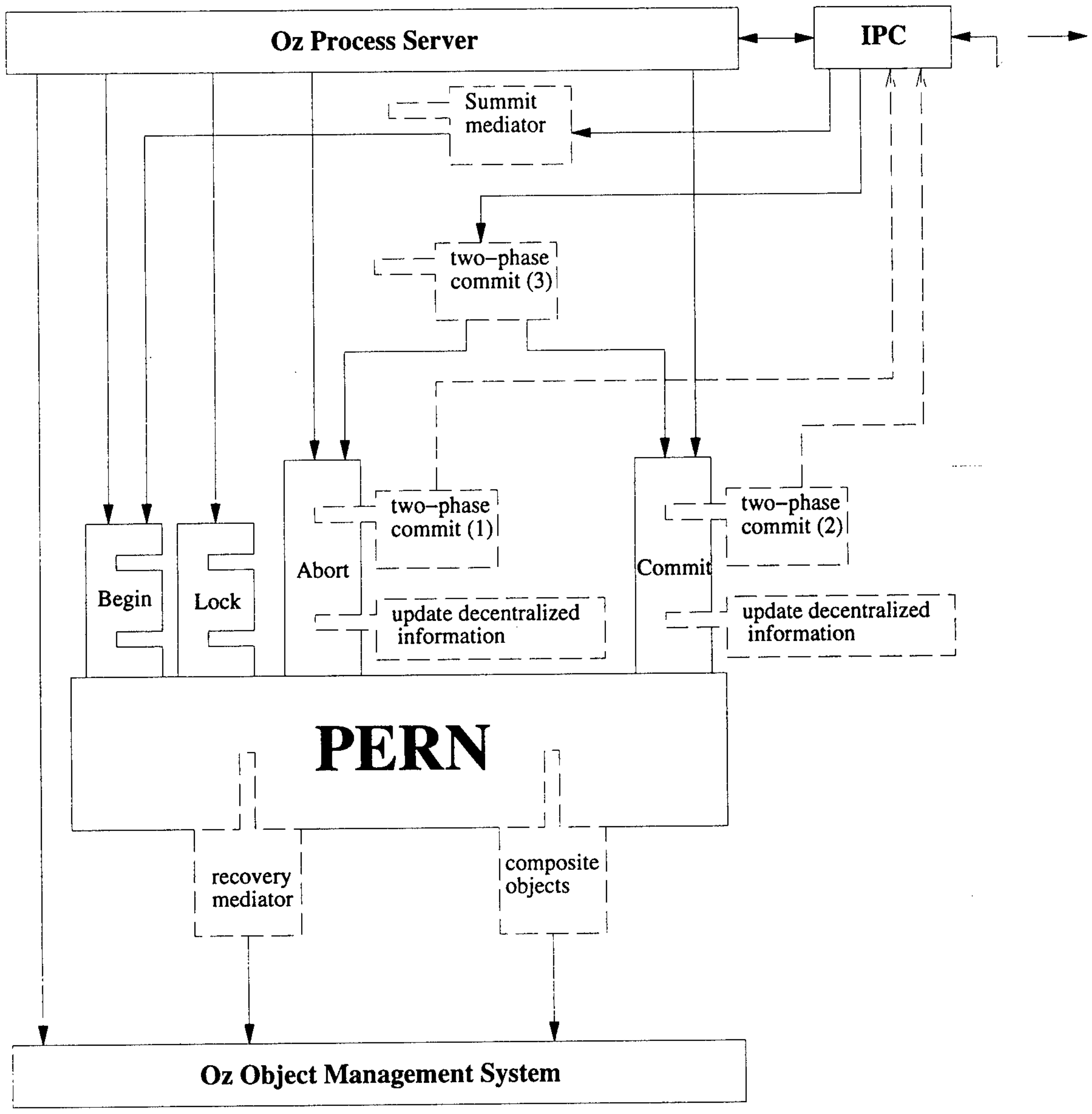


Figure 10.6: Interfacing PERN with OZ

server. PERN 's mediators maintain the necessary decentralized information that allows a collection of PERN instances to work together to fulfill the requirements of executing a Summit.

Transaction semantics for Summits can be divided into two parts: synchronizing the updates by a process-step across several sites (horizontal) and guaranteeing consistency across process-steps (vertical). The approach to Summit transactions must not greatly affect the processing of local, independent transactions at each site. Recall that each site, by default, has complete autonomy with respect to all other sites, and any relaxation of autonomy must be agreed to *a priori* by all parties. Each PERN instance, therefore, determines whether to permit or deny data access from local process-steps. When a site is involved in a Summit, however, it must necessarily give up some autonomy in order to collaborate with other sites. The site which starts a Summit is called the *coordinator;* the other participating sites are called *participants.* The Summit transaction protocol (not to be confused with the execution protocol) follows these rules:

1. A Summit process-step, by definition, accesses data from multiple sites; therefore a local transaction is needed at each site for each Summit process-step.
2. A participant site cannot commit the local transaction it created for a Summit until the coordinator directs it to.
3. The coordinator cannot commit its transaction for a Summit process-step until all atomicity requirements have been fulfilled at all sites.

The mediators in Figure 10.6, shown here for a PERN instance at a single site, maintain the appropriate inter-site transaction dependencies. When the coordinating site wishes to complete a Summit, the `two-phase commit` mediator (number (2) in Figure 10.6) is invoked (since it is the *before* mediator for commit). This mediator initiates a two-phase commit protocol with all sites involved in the Summit by sending messages to the remote sites through the IPC layer. The remote `two-phase commit` mediators (number (3) in Figure 10.6)) receive the request and commit their transactions, thus guaranteeing consistency at all sites involved in the Summit.

10.6 Lessons Learned

The ProcessWEAVER demonstration covered only very simple cases, such as the example in Section 10.4. This was a necessary first step, however, since ProcessWEAVER itself has no notion of concurrency control at all. Now that we have a structure that employs ECC to support conventional transactions, we are ready to move on to cooperative transactions for software development processes.

After completing this pilot study with ProcessWEAVER we removed the existing transaction manager from OZ and substituted PERN in its place. Since PERN is descended from MARVEL 's transaction manager module, we expected an easy integration. We were pleasantly surprised at how easy it was to support decentralized transaction management using multiple instances of a centralized transaction manager; the mediator architecture made this possible and reduced the need to develop duplicate functionality. For example, there was no need to implement distributed deadlock prevention since the mediators were able to reuse the communication deadlock logic from the OZ process servers. The OZ system is composed of roughly 200,000 lines of C code; the PERN component is 13,000 lines of C while the PERN /OZ mediators are 1,800 lines.

The basic architecture in Figures 10.4 and 10.6 is the same. If OZ had not had "hooks" for transactions, such as the `consistency` annotations inherited from MARVEL [14], the details of integrating PERN with OZ and with ProcessWEAVER would also have been very similar. For example, we could have incorporated PERN via OZ 's enveloping mechanism [86] so that when OZ invoked an activity, PERN would intervene and first lock all parameters to the activity. When the activity completed, PERN would release the locks. Such an implementation would map each activity to one transaction. Multiple process-steps could be grouped into transactions by adding "dummy" process-steps, with appropriate prerequisites and consequences, to begin and end the transactions. The mediators would be different, of course, but the underlying architecture would remain the same.

10.7 Contributions and Future Work

Integrating pre-existing, independently developed components is far from straightforward; the architecture presented in this chapter shows how mediators can aid this effort. In the absence of standard interfaces for components, particularly legacy systems, mediators can provide the necessary veneer to allow components to work together and/or be added onto "complete" systems.

Our main contributions are:

- Requirements for a concurrency control component that exists separately from task management services in SDEs.
- A general architecture for integrating task management services and an external concurrency control component.
- A sample external concurrency control component, PERN .
- Successful application of our architecture to add an external concurrency control component onto a commercial product, ProcessWEAVER, with no prior notion of concurrency control.

There are many avenues for future work. The most pressing problem is dealing with environment frameworks with their own built-in concurrency control policy that is nonetheless deemed unsuitable for some potential framework applications, such as computer-supported cooperative work. A sophisticated mediator that introduces concurrency control or overrides an existing mechanism (like conventional transactions or the checkout model) cannot operate if there is no way to insert effective entry points and callbacks to/from the mediator. Fortunately environment frameworks already provide some facility for invoking external tools, making a standard architecture with application-specific mediators plausible. In this chapter we have minimized crash recovery (as opposed to rollback for concurrency control purposes), but there are complex issues regarding recovery in tandem with cooperative transactions [102].

Chapter 11

Extended Transaction Modeling

Abstract

Database management systems (DBMSs) have been increasingly used for advanced application domains, such as software development environments, workflow management systems, computer-aided design and manufacturing, and managed healthcare. In these domains, the standard correctness model of serializability is often too restrictive. We introduce the notion of a Concurrency Control Language (CCL) that allows a database application designer to specify concurrency control policies to tailor the behavior of a transaction manager. A well-crafted set of policies defines an extended transaction model. The necessary semantic information required by the CCL run-time engine is extracted from a *task manager*, a (logical) module by definition included in all advanced applications. This module stores task models that encode the semantic information about the transactions submitted to the DBMS. We have designed a rule-based CCL, called CORD, and have implemented a run-time engine that can be hooked to a conventional transaction manager to implement the sophisticated concurrency control required by advanced database applications. We present an architecture for systems based on CORD and describe how we integrated the CORD engine with the Exodus Storage Manager to implement Altruistic Locking.

11.1 Introduction

Advanced database applications (henceforth applications) require more sophisticated concurrency control mechanisms than the standard ACID transaction model provides [16, 165, 50]. For this reason, many *extended transaction models* (ETMs) have been developed [75, 199, 128] that rely on special *semantic information* about the transactions and their operations. There is no consensus, however, as to which ETM is appropriate for advanced applications; most likely, there never will be, since each ETM is optimized for a particular behavior. Therefore, a database management system (DBMS) cannot implement an ETM suitable for all applications. One possible goal is to design a DBMS whose transaction manager (TM) can be tailored to provide the desired ETM for a given application. An even better direction is to show how to extend the TM for existing DBMSs to provide such ability.

Advanced applications include software development environments, network management, workflow management systems, computer-aided design and manufacturing, and managed healthcare. These diverse applications have one feature in common – they have a task manager that stores rich semantic information about the transactions submitted to DBMS. The FlowMark Workflow system [150], for example, stores a workflow process as a directed acyclic graph of activities. In the Oz Process-Centered Environment [21], a process engine interprets task models encoded in planning-style rules. Since the actual implementation of the task manager changes from one application to the next, we do not present details for any particular task manager, nor do we cover situations where the semantic information is implicit and/or arbitrarily spread across multiple parts of the application. We also focus our attention on concurrency control, rather than recovery issues.

Extensible concurrency control is the ability for the TM of a DBMS to alter its decisions regarding how transactions are allowed to behave. It is commonly accepted that semantic information about the transactions is necessary to realize extensible concurrency control. This chapter investigates how the TM can acquire semantic information from the task manager of an application, and how to flexibly direct TM to incorporate this information when making concurrency control decisions. This research is performed in the context of showing how to augment existing TMs to support the necessary advanced transaction behavior.

In an application for a large bank, for example, a user's Withdrawal transaction should not be forced to wait while the bank runs a long-duration Balance transaction. The bank application designers could directly modify the existing TM of their DBMS (including rewriting it) to implement the behavior in Figure 11.1a, but this effort would be costly and have to be repeated for each such scenario. Alternatively, the application could be tightly integrated with the TM (e.g., transactional workflows [81]), granting the application fine-grained control over transaction behavior. The original reason for introducing transactions, however, was to avoid such solutions that often reduce to low-level concurrent programming; also for practical reasons, the application and TM should remain separate entities.

Motivating Example

Consider solving this banking example to allow the Balance transaction to observe temporarily inconsistent data. If the TM has a sophisticated interface, such as Encina [225], it might be possible to modify and reimplement the application for an individual case. As more and more special cases arise, however, some model is needed to reduce complexity; as an example, Epsilon Serializability [187] (**ESR**) is an ETM that increases concurrency by allowing bounded inconsistencies to occur. The TM could be reimplemented to support **ESR**, but if the behavior changed yet again, more reimplementation would be necessary. The goal of our research is to provide a solution whereby the application designer need only produce a specification, such as the simplified **ESR** example in Figure 11.1b, that tailors the behavior of the TM.

This chapter introduces a component used by the TM to tailor its behavior based upon an ETM specification written in a concurrency control language (CCL). Because supporting an ETM requires semantic information from the application, this component employs a generic interface

```
if (Conflict between Balance and Withdrawal) then
    if (Total-Off + Withdrawal Amount < 1 Million) then
        Total-Off += Withdrawal Amount
        Ignore Conflict
    fi
fi
```

(1a) Simple Case for Bank Policy

```
if (Read/Write conflict) then
    if (can tolerate increase in inconsistency) then
        Update inconsistency totals
        Ignore conflict
    else
        Abort conflicting transaction
    fi
else if (Write/Write conflict) then
    Abort conflicting transaction
fi
```

(1b) Simplified **ESR** CCL specification

Figure 11.1: Concurrency specifications

to *extract* the semantic information; a *mediator* layer of special-purpose code insulates the CCL engine from the application. An application designer can thus extend a TM by providing an ETM specification and mediator code, as needed, to extract the necessary semantic information from the target application. We envision that such CCL engines can be attached to existing DBMSs (with only slight modifications to the DBMS) to provide immediate extensibility.

The basic building block of an ETM is a *concurrency control policy* (henceforth, policy) that defines how a TM should react to non-serializable access exhibited by two conflicting transactions. The ETM specification enumerates the differences from serializability, the standard correctness model for most DBMSs. We view approaches that model every database access by all transactions (such as pattern machines [208] or Relative Serializability [4]) as impractical since each transaction that wishes to relax atomicity would first have to analyze the operations of all other potentially affected transactions.

We present the features of a rule-based CCL called CORD (for COoRDination) and its CORD engine, and show how to implement Altruistic Locking (**AL**) [199], a well-known ETM from the literature. We then discuss our experience integrating the CORD engine with the Exodus Storage Manager [38] to implement **AL**. Finally, we evaluate our efforts and related work, and summarize our contributions.

11.2 Architecture

Figure 11.2 shows the integration of a CCL engine into a DBMS used by an application. A well-defined task manager module stores semantic information about the transactions submitted to the TM. The ETM specification is first translated into a machine-readable format that is loaded by the CCL engine upon initialization. We assume that the DBMS is dedicated entirely for use by the application. All operations submitted to the TM that remain serializable are processed without invoking the CCL engine. When serializability conflicts occur, the TM invokes the CCL engine to locate a policy (if any) that matches the observed conflict. The CCL engine employs a generic interface to extract semantic information from the application using special mediator functions provided by the application designer (shown in dashed boxes).

The CCL engine places certain requirements on the TM, which we assume already has a well-defined API of primitive operations, such as Begin and Commit. First, we must modify the TM to invoke the CCL engine when it detects a conflict (typically by modifying the TM's lock manager). Second, we need `before-` and `after-` callback functions for each API operation so that the CCL engine can alter and extend the functionality of the TM. When the TM is requested to lock an

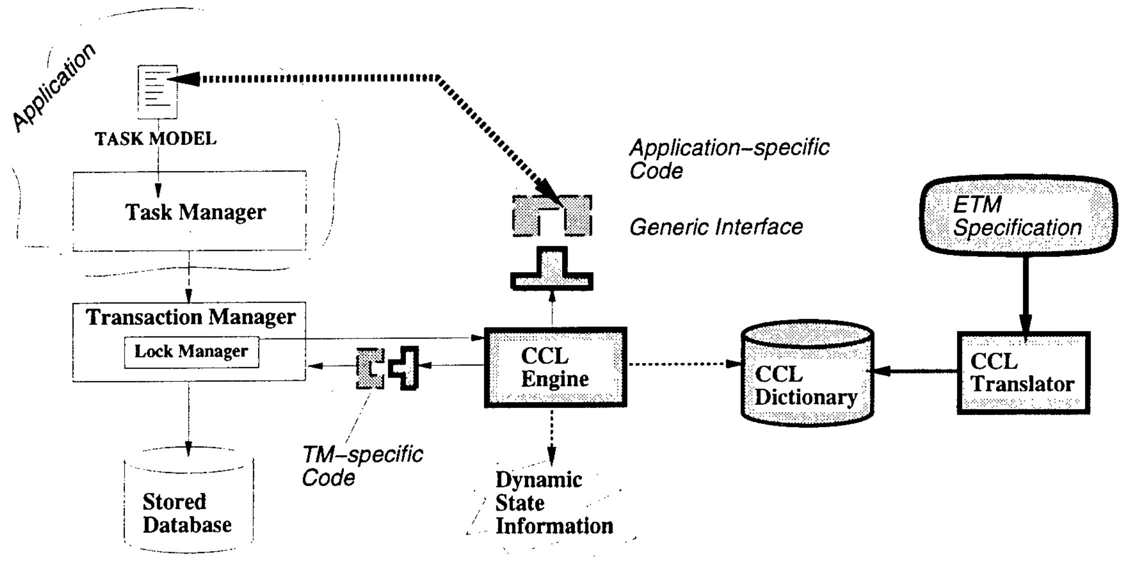


Figure 11.2: CCL extension to DBMS architecture

object, for example, the `lock_before` callback can invoke a mediator function to determine if the ETM allows the transaction to access the desired objects, and possibly deny the primitive operation. Similarly, a `lock_after` callback can trigger other actions as required by the ETM. These changes are represented by the thin black rectangles in Figure 11.3; Figure 11.4 shows the modified **Lock** primitive in more detail. We feel these two features should be part of any DBMS that provides extensible transaction management; in fact, this interface is already very similar to Encina [225]. In Section 11.5 we describe how we modified the Exodus Storage Manager to include these features. Our success at being able to modify a foreign system leads us to believe that DBMS designers themselves would be able to modify their systems accordingly.

11.3 ETM Specification

A CORD ETM specification contains a preamble and a set of CORD rules that the CORD engine loads when initialized by the TM. The CORD language is a rule-based CCL influenced by Barghouti's Control Rule Language [13]. The preamble defines mediator functions in an objectcode file that will be dynamically linked with the CORD engine to extend its functionality. Each CORD rule consists of a sequence of policies defined as condition/action pairs that tell the TM how it should behave under certain circumstances; CORD rules are thus similar to planning-style rules.

When a conflict is detected between two accesses to an object, the TM invokes the CORD engine to resolve the conflict, constructing a *scenario* containing the object's unique identifier (oid) and class name, and the unique transaction identifiers (tid) of the two conflicting transactions (i.e., if three transactions conflict with each other, the conflicts are handled in pairwise fashion; [102] presents an approach for handling sets of conflicts at once). The CORD engine acts like an expert system, reacting to conflicts by invoking the appropriate policy. If no suitable policy is found, the TM responds to the conflict in its usual fashion.

The CORD language defines an extensible set of data types to model the dynamic state information needed by the policies. The standard data types, shown in Figure 11.5, model information from the TM: *transaction*, *object*, and *lock*. For clarity of presentation, we assume the TMs are lock-based. For a given transaction T_{17}, for example, the TM may keep a large data structure storing log records, lock sets, and other pertinent information. The CORD engine maintains its own dynamic state information about T_{17}, separate from the TM, by instantiating an object from its transaction type;

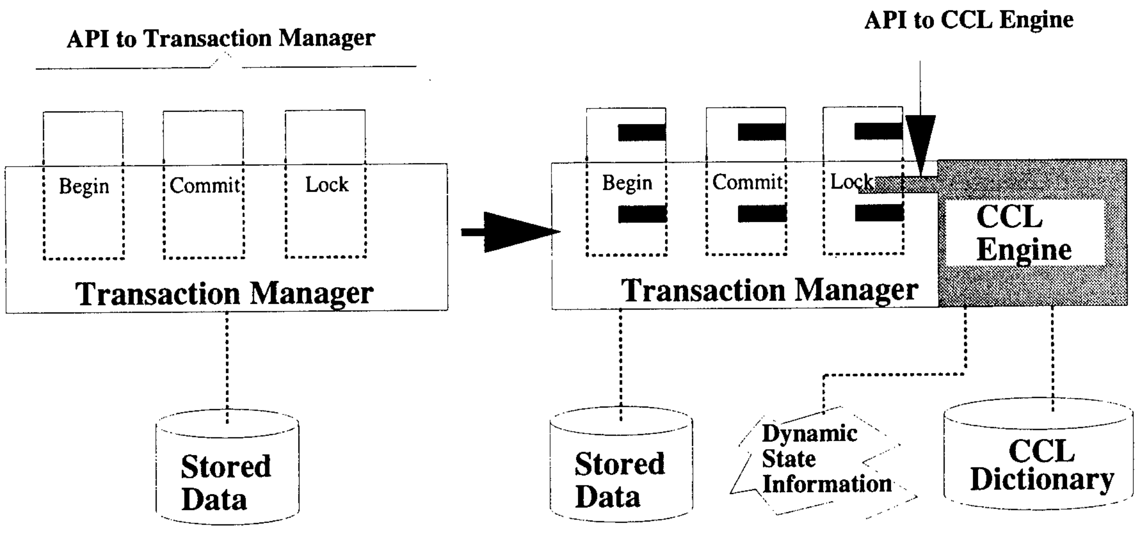


Figure 11.3: CCL engine integrated with Transaction Manager

```
forward function resolve_conflict (in obj_list, out resolved, out info) : lock_list
type service_status = (SERVICE_OK, SERVICE_DENY, SERVICE_OVERRIDE)

function Lock(in t, in obj_list, in mode) : boolean
    rc := lock_before(t, obj_list, mode);
    if (rc = SERVICE_OVERRIDE) then return (true); fi
    if (rc = SERVICE_DENY) then return (false); fi
    if Lock can be granted then
        Normal transaction behavior
    else
        ⇒Interface to CCL Engine
        CS := construct_scenario (t, obj_list, mode);
        locks := resolve_conflict (CS, resolved, cord_info);
        if (resolved = false) then return (false); fi
    fi
    lock_after (t, obj_list, mode);
    return (true);
end
```

Figure 11.4: Modified **Lock**$(t, obj\_list, mode)$

type	*attribute*
transaction	tid, lockset, parent, subtransactions, top_level
object	oid, name, lockset, class
lock	lock_mode, tid, object

Figure 11.5: Default CORD semantics

since the *tid* is the same, the engine can communicate with the TM to extract detailed information about, and perform actions on, transactions. These data types can be extended to store additional information needed to support a set of policies. The ETM specification, for example, might request the CORD engine to store additional task information in a `task` attribute for each transaction type.

Each CORD rule is parameterized by the class of object (within the DM) to which it applies, since conflicts occur on individual objects (ENTITY applies to all classes). These CORD rules can be viewed as concurrency control *methods* that an object employs to resolve conflicts (similar to an approach suggested by [92]). Multiple rules defined for the same class are differentiated by a *selection criterion*, allowing the CORD engine to select at run-time the most applicable CORD rule.

Each CORD rule can optionally bind variables (shown as `?var`) that refer to semantic information required by its policies. There are five default variables describing the conflict scenario: `?ConflictObject`, `?Lconflict` (i.e., the conflicting lock being requested), `?Tconflict`, `?Lactive` (i.e., the existing lock), `?Tactive`. For example, if a policy needs to refer to the parent of the transaction causing the conflict, the following variable would be defined within the CORD rule: `?Tpar = ?Tconflict.parent`.

The condition for a policy specifies logical expressions on the rule's variables to determine which one is valid. It can perform simple comparisons of attribute values, such as checking whether the lock mode requested by the active transaction is in read mode. The CORD engine can dynamically load in new code, as determined by the ETM specification, to introduce new functions used when evaluating these conditions; as we will see, this is a powerful mechanism.

Most TMs can only suspend or abort a transaction to resolve serializability conflicts. In contrast, CORD policies can perform arbitrary actions on transactions as needed to implement a particular ETM. There are two ways that a conflict can be resolved: first, it can be ignored, because it is only a serializability conflict, not a conflict according to the specified ETM; second, the TM can take action, such as suspending or aborting transactions, creating dependencies between transactions to maintain integrity, or dynamically restructuring transactions. In addition to CORD's default actions, described next, new actions can be implemented and dynamically linked with the CORD engine.

The most basic CORD action, `ignore()`, allows non-serializable accesses as directed by the ETM. This action, for example, allows transactions to share partial results with one another or commuting operations to be performed. If there are side-effects of the non-serializable accesses (as determined by the ETM designer), then the CORD engine can maintain *dependencies* between the conflicting transactions. The CORD language allows commit and abort dependencies to be formed between transactions. Briefly, if T_i has an abort dependency on T_j, then if T_j aborts, T_i must also abort. If T_i has a commit dependency on T_j, then T_i cannot commit until T_j finishes (either commits or aborts). The `add_dependency (?Tconflict, ?Tactive, abort)` action, for example, ensures that if the TM ever aborts `?Tactive`, the CORD engine will abort `?Tconflict`. Removing dependencies between transactions, for example, allows a sub-transaction to be treated as top-level. ACTA [42] defines twelve types of dependencies between transactions, but we limit CORD to these two since most of the ACTA dependencies are the domain of the database application, and too tightly bind the TM with the task manager. `suspend (?t1, ?t2)` blocks transaction `?t1` until `?t2` has either committed or aborted, `abort (?t)` aborts a particular transaction, while `notify (?t1, msg)` action delivers a message to the application on behalf of transaction `?t1`. Other actions are defined in [102].

The DBMS engineers are responsible for integrating the TM with the CORD engine. In addition to the effort outlined in Section 11.2, this means that mediator functions need to be written for each of CORD's default actions to interface to the specific TM. For example, the CORD engine must map its **suspend** action to specific capabilities in the TM. Examples of these mediators can be found in [106]. The mediator for the `notify` CORD action is written by the application designer to interface TM with the application. Through such mediation, the CORD engine is insulated from the details of the other system components.

11.3.1 Motivating Example Revisited

To return to our opening example, we now present a CORD implementation of **ESR** [187]. Each Epsilon Transaction (ET) has a specification (called an ϵ-spec) of its allowed *import* and *export* inconsistency. A transaction imports inconsistency by reading the uncommitted results of an update transaction; this update transaction is then considered to have exported inconsistency. Separate from these ϵ-spec values, each data item has its own data-ϵ-spec for the amount of inconsistency it allows. Note that **ESR** is equivalent to Serializability if all transaction-ϵ-spec values are 0. In this chapter, we implement a restricted form of **ESR** that does not allow update transactions to import inconsistency; a more general form of **ESR** has been implemented in [102].

Each ET maintains a fixed *ImpLimit* (*ExpLimit*) as part of its ϵ-spec that determines the bounded amount of inconsistency it can import (export). Each ET also maintains a running import (export) accumulator that it updates whenever it imports (exports) inconsistency. When an ET attempts to read and write a data item x, the inconsistency inherent in x is added to the ET's inconsistency counters. Each data item maintains an accumulator of inconsistency used to check against its data-ϵ-spec. Summing up, the CORD engine must store four new pieces of information with each ET (*ImpLimit, ExpLimit, import_accumu, export_accumu*) and two pieces of information with each data item (*data-ϵ-spec* and *data_accumu*). We assume here that the data-ϵ-spec value is stored in the DBMS itself (as an attribute for each object).

The CORD engine maintains this state information about ETs and enforces the inconsistency limits. Using callback functions, the task manager (requesting the locks) can be queried to find out how much each ET will alter the data item's value. For example, **ESR::lock_after** in Figure 11.7 is invoked to create a CORD data structure of type ESR_accumu to store the inconsistency introduced for each object as ETs proceed. The `DBMS::` functions retrieve the desired information using the API of the underlying DBMS.

To complete our **ESR** implementation, the CORD rule (in Figure 11.6) contains four policies to allow a query ET and an update ET to conflict if the transaction ϵ-spec values of the involved ETs are satisfied. The mediator functions referenced in this CORD rules are dynamically loaded from `esr.so` and can be found in [106]. The TM does not need to be reimplemented to support the **ESR** behavior; only special-purpose mediator code needs to be written that extends its behavior as desired.

11.3.2 Extracting Semantics

The novel feature of the CORD language is that it allows the application designer to model the desired semantic information in the application. For each piece of semantic information, an *access* mediator is implemented (by the application designer) that will extract the information at run-time if needed by the CORD engine. At startup, the CORD engine dynamically loads in the code for the access mediators from the ETM specification. The CORD engine employs a generic mediator interface to extract the desired semantic information (as shown in Figure 11.3). If either the application or the TM is replaced, only the specific mediator functions need to be rewritten; the CORD engine remains unchanged.

11.4 Example Extended Transaction Model

We now present a full CORD solution to extending a TM for Altruistic Locking (**AL**) ETM [199]. **AL** is an extension to two-phase locking (**2PL**) [62] that accommodates long-lived transactions.

```
# Preamble
cord_rules
    object_code esr.so
    condition   valid_tolerance (object, transaction, transaction)
    action      increment_accumu (object, transaction, transaction)

# Rules
epsilon_extension [ ESR_ENTITY ]
    selection_criterion:
        lock.lock_mode: W, R
    body:
        # Conflict between an update (that requests the lock) and
        # a query (that has the lock). Verify that the resulting
        # increase in inconsistency will be tolerated.
        if (and (?Lconflict.lock_mode = W)
                (valid_tolerance(?ConflictObject, ?Tconflict, ?Tactive))) {
            increment_accumu(?ConflictObject, ?Tconflict,?Tactive)
            ignore()
        }

        # Conflict between a query and an update (that has lock)
        if (and (?Lactive.lock_mode = W)
                (valid_tolerance(?ConflictObject, ?Tactive, ?Tconflict))) {
            increment_accumu(?ConflictObject, ?Tactive, ?Tconflict)
            ignore()
        }

        # The following conditions match when inconsistency is too much
        if (?Lconflict.lock_mode = W) { abort(?Tconflict) }
        if (?Lactive.lock_mode = W) { suspend(?Tconflict, ?Tactive) }
    end_body
```

Figure 11.6: CORD rule for Epsilon Serializability

```
procedure ESR::lock_after (in t, in obj_list, in mode)
    for oid in obj_list do
        if (DBMS::get_att_value (o, "d_espec", d_espec)) then
            DBMS::get_att_value (o, "value", d_value);
            if (not ESR_globals::member (oid)) then
                one = ESR_accumu::new (obj_id:oid,
                    consistent_value:d_value, espec:d_espec, accumu: 0);
                ESR_globals::insert (one);
            fi
        fi
end
```

Figure 11.7: ESR::lock_after mediator algorithm

AL1	Two transactions may not simultaneously hold *conflicting* locks on the same object unless one of the transactions first donates the object.
AL2	If T_a is *indebted* to T_b, then it must be completely in the wake of T_b until T_b performs an Unlock.
$d(a)$	Transactions that have donated, but not unlocked, a.
$in(a)$	Transactions that readers of a must be in the wake of.
$W(T)$	Transactions whose wakes T is completely within.
$J(T)$	Transactions whose wakes T <u>must</u> be completely within (based on *AL1* and *AL2*).

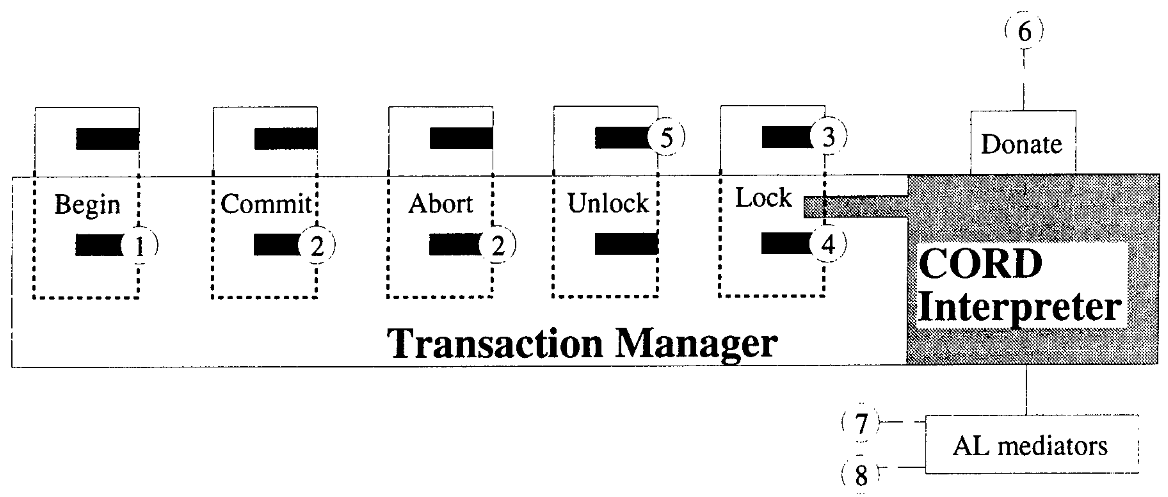
Figure 11.8: **AL** requirements

Under **2PL**, short transactions will encounter serious delays since database resources can be locked for significant lengths of time. In **AL**, several transactions can hold conflicting locks on a data item if constraints *AL1* and *AL2* in Figure 11.8 are satisfied. In this example, read and write locks have the usual semantics. Using the *Donate* operation – a new TM primitive operation – a transaction announces to the database that it will no longer access a given data item, thus allowing other transactions to access it (constraint *AL1*). A donate is not an unlock and the transaction must still explicitly unlock data items that it has donated – the transaction is free to continue locking data items even after some have been donated.

A transaction enters the *wake* of transaction T_i when it locks an object that has been donated (and not yet unlocked) by T_i. A transaction is *completely* in the wake of T_i if all the objects it locks are donated by T_i. If T_j locks an object that has been donated by T_i, T_j is *indebted* to T_i if and only if the locks conflict or an intervening lock by a third transaction T_k conflicts with both. Even though two read locks are compatible, the second read becomes indebted to the first when an intervening write occurs. Initially, for all a and T, $J(T) = d(a) = in(a) = \emptyset$. By default, as each transaction begins, it enters the wake of all active transactions; elements are removed and inserted into $W(T)$ based upon the behavior of T.

The CORD engine maintains dynamic state information about the wakes of transactions (i.e., $W(T)$ and $J(T)$) and enforces the indebted constraint *AL2*. $W(T)$ is calculated by tracking the set of active transactions with the `begin_after` mediator **AL::begin_after**. We extend the **Lock**(T, *obj_list*, *mode*) primitive operation (shown in Figure 11.4) by binding the `lock_before` mediator to **AL::lock_before**. The $J(T)$ and $W(T)$ sets are updated by the `lock_after` mediator, **AL::lock_after**. These sets cannot be updated in the `lock_before` mediator, otherwise a locking conflict that failed to set a lock would incorrectly update this information. The **Unlock** operation is extended by binding its `unlock_before` mediator to **AL::unlock_before**; this mediator and the new **AL::donate** primitive operation manage $d(a)$. When transactions commit (abort), the `commit_after` (`abort_after`) mediator, bound to **AL::complete_tx**, updates *ActiveSet*. The **AL** protocol presented in [199] upgrades read locks to write locks solely to preserve the indebted relationship between transactions. Instead of altering the locks held by the lock manager, our solution maintains several sets for each database object a and transaction T, as shown in Figure 11.8.

Our implementation is completed by two CORD rules. The `AL_rw` CORD rule in Figure 11.10 is invoked for all read/write conflicts on any object. The policy of this rule handles situations when a write lock is requested on an object that `?Tactive` previously read and donated; this carefully maintains the indebted relation, *AL2*. The policy in `AL_ww` allows multiple writers if the conflicting object was donated first. Figure 11.9 shows the interaction between TM and the CORD engine. As transactions request primitive operations from the TM's API, the various mediator functions (encapsulated by ovals) are invoked through callbacks. When the CORD engine evaluates its policies,



```
(1) procedure AL::begin_after (in $T$)
    $W(T) := ActiveSet$;
    $ActiveSet := ActiveSet \cup \{T\}$;
end

(2) procedure AL::complete_tx (in $T$)
    // T can no longer impact any transaction, so update the appropriate W(t) sets.
    $ActiveSet := ActiveSet - \{T\}$;
    foreach $t \in ActiveSet$ do
        if $(T \in W(t))$ then $W(t) := W(t) - \{T\}$;
end

(3) function AL::lock_before (in $T$, in $a$, in mode) : service_status
    // If a hasn't been donated, SERVICE_OK is returned if T is not completely within the
    // wake of another transaction. Otherwise, SERVICE_OK is returned if T remains
    // completely in the wake of J(T).
    $w := W(T) \cap d(a)$;
    $i := J(T) \cup in(a)$;
    if $(i \subseteq w)$ then return (SERVICE_OK); fi
    return (SERVICE_DENY);
end

(4) procedure AL::lock_after (in $T$, in $a$, in mode)
    // Now that lock is assured, we update the appropriate sets. Can only do here.
    $J(T) := J(T) \cup in(a)$;
    $W(T) := W(T) \cap d(a)$;
    return (SERVICE_OK);
end

(5) function AL::unlock_before (in $T$, in $a$, in mode) : service_status
    // Removes downstream transactions from wake of T, and maintains in(a) and d(a).
    $d(a) := d(a) - \{T\}$;
    $in(a) := in(a) - \{T\}$;
    foreach $t \in ActiveSet$ do
        if $(T \in J(t))$ then $J(t) := J(t) - \{T\}$;
    return (SERVICE_OK);
end

(6) procedure AL::donate (in $T$, in $a$)
    // New operation in DBMS API
    $d(a) := d(a) \cup \{T\}$;
end

(7) function AL::is_donated (in $a$) : boolean
    // New CORD condition
    return (not $(d(a))$);
end

(8)procedure AL::update_in_set (in $a$, in $L$)
    // New CORD action
    if $(L.lock\_mode = R)$ then $in(a) := in(a) \cup \{ L.tid \}$; fi
end
```

Figure 11.9: Mediator extensions for **AL**

```
# Preamble
cord_rules
    object_code   alt.so
    condition     is_donated (object)             # New AL conditions
    action        update_in_set (object,lock)     # New AL actions

# Rules
AL_rw [ ENTITY ]
    selection_criterion: lock.lock_mode: W, R
    body:
        # Allow conflict on donated object; maintain indebted relationship
        if (is_donated (?ConflictObject)) {
            add_dependency(?Tconflict, ?Tactive, abort)
            update_in_set(?ConflictObject,?Lactive)
            ignore()
        }
    end_body

AL_ww [ ENTITY ]
    selection_criterion: lock.lock_mode: W, W
    body:
        # Allow conflict on donated object
        if (is_donated (?ConflictObject)) {
            add_dependency(?Tconflict, ?Tactive, abort)
            ignore()
        }
    end_body
```

Figure 11.10: CORD rule to support **AL**

it employs the new **AL::is_donated** condition and **AL::update_in_set** action as defined in the preamble. Neither the TM nor the CORD engine were altered in any way; only the ETM specification and mediator code (included dynamic code) were added for this solution.

11.5 Example Integration with DBMS

We next integrated the CORD engine with the Exodus Storage Manager [38]. In Exodus, client applications share memory pages from a virtual "volume" residing on a storage manager server. A locking conflict, therefore, occurs on a particular page and volume. Objects in Exodus can be small enough to fit several to a page, or one object can be spread across many pages. Client applications requests objects from the server by page location.

We modified the Exodus lock manager (LM) to invoke the CORD engine when it detects a lock conflict. The CORD engine then uses the available semantics (in this case, only the lock modes being requested) to determine if a CORD rule matches the conflict scenario. LM had twelve individual locations where it checks a lock matrix to determine if two lock modes are compatible; for example, `if (!LM_Compat[lockEntry.lockMode][requestMode])` was replaced with:

```
if (!cord_compatible(lockEntry.headerList.transRec, lockEntry.lockMode,
    transRec, requestMode, lockEntry.lockHeader))
```

The extra parameters refer to the actual transaction structures in Exodus. Five such expressions were replaced with calls to `cord_compatible()`, five others were changed to call a similar function, `cord_compatible_upgrade()`, used when the client upgrades its lock; the last two points were changed to `cord_compatible_list()`; these functions may be found in [106]. To complete the infrastructure, we augmented the interface for each primitive operation in the transaction API to allow callback functions. At various points in Exodus, we inserted `before-` and `after-` mediator calls:

```
if (lock_before (transRec, lockid.page.pid.page, requestMode) != 1)
```

ETM	CORD rules	Number of Mediators
strict **ESR** in Oz	1	8 (325 lines)
general **ESR** in Oz	2	10 (446 lines)
AL in Exodus	2	10 (429 lines)

Figure 11.11: Statistics on implementing ETMs

```
return (esmFAILURE);
```

11.5.1 Altruistic Locking in Exodus

To complete the **AL** implementation in Exodus, we extended the communication protocol between the client and server to provide a new *DonatePage* operation. Exodus allows client programs to scan through a collection of objects (called a file) in the storage manager, guaranteeing that all objects in the file will be accessed exactly once during the traversal. Since several objects can reside on a page, we need to be careful to donate a page only when it is no longer needed by the client: whenever the scan iterator retrieves a new page from the server, the client application calls a new Exodus operation, *DonatePage*, directing the server to donate the previous page which will no longer be used by that client. When the scan iterator is complete, the client donates the last page in the scan before committing. In conjunction with this new function, the same CORD rule from Figure 11.10 is used to allows particular behavior in the TM.

We encountered two "feature interaction" problems. The first, which we call the *double-buffering* problem, occurs when a client donates a page it has updated. Recall from Section 11.4 that under **AL**, the transaction that donates a page does not unlock it. If the transaction only flushes its pages onto durable storage when it commits, future transactions that read this page will see the old value. This problem could have been foreseen since **AL** does not guarantee failure-atomic transactions and Exodus implements log-based recovery based on the ARIES algorithm [163]. We therefore need to flush these pages to storage whenever a page is donated. Also, since the client forces pages to the server when a transaction commits, we must not force already donated pages. Thus, implementing the Donate operation itself required some effort. In [199], the authors discuss other problems, related to recovery, that **AL** might introduce. The second problem reveals a subtle interaction between granularity locking [88] and **AL**. When scanning a file, Exodus acquires a file lock instead of acquiring a separate lock for each page in the file. The transaction scanning a file, however, cannot donate this file lock until it completes the scan. Instead, we programmed a client application to mimic a scan by manually requesting each page in order.

11.6 Evaluation

In the previous chapter, we presented an architecture for integrating a TM component into environment frameworks. We described how a mediator architecture allowed us to implement distributed two-phase commit on top of a set of centralized (but extensible) TMs. In this chapter, we have shown how to extend a TM to tailor its behavior using our CORD engine. The statistics shown in Figure 11.11 summarize our effort to use the CORD engine to support a particular ETM. The size and number of mediator code is reasonable, especially considering the extra benefits of being able to tailor an ETM on top of an existing DBMS. The **ESR** CORD experiment was implemented and tested within Oz [21], a rule-based process-centered environment. The **AL** solution was first designed and tested for a small demonstration environment and then was reproduced within Exodus. The same CORD engine was used for all experiments: only the mediator code and ETM specifications changed.

11.6.1 Support for Locking

The TM can detect serializability conflicts using either locking, timestamp ordering (**TO**) [32], optimistic concurrency control (**OCC**) [141], or any other equivalent method. In addition to being the most popular, we feel that locking is most suitable for extensible concurrency control. **OCC** is inappropriate for several reasons. First, **OCC** determines conflicts after they occur, when a transaction, T, attempts to commit. If T conflicts with a previous transaction, T_c, that has already committed, it might not be possible for the TM to extract any semantic information about T_c, since the information for the task that employed T_c might no longer be available. Second, if negotiation were used to resolve the conflict, such interaction must occur as the conflict occurs, not (possibly) long after the transactions conflict. A timestamp-based protocol would not be as efficient, either. If the CORD engine needs to inspect the objects a transaction has accessed, for example, locking already provides lock sets for each transaction, but there is no similar concept in **TO**; the CORD engine would have to duplicate this information. As much as possible, we want the CORD engine to only maintain dynamic state information that is not already managed by the TM. One limitation of our approach is that it does require changes within the TM, but the API changes are minimal and in-line with standard APIs (as in Encina).

11.6.2 Effects on application

The CORD actions necessarily affect the advanced database application. In client/server architectures, the client typically waits synchronously for a reply from the server; to suspend a transaction, the TM can simply delay its response. If the TM is bundled together with the database application in one single-threaded operating system process (for example, a workflow engine combined with a database), suspending a transaction is not easy at all, since multiple contexts need to be carefully maintained and restarted at the correct times. In the context of Barghouti's Control Rule Language, we successfully implemented the CORD **suspend** action in the MARVEL process-centered environment [31], but this required significant portions of process engine (i.e., MARVEL 's task manager) to be reimplemented to be aware that a process task could be suspended during its execution. General solutions to the problem of how applications should react to ETMs are outside the scope of this report.

The actions in the CORD language must be matched to the capabilities present in the TM and the task manager. When attaching the CORD engine to an existing TM, the CORD primitive actions (i.e., abort, suspend) are parameterized to invoke corresponding primitives from the API for the TM. If the TM cannot suspend transactions, for example, no CORD rule can use this primitive.

11.7 Related Work

The ACTA framework [42] constructs a theoretical model that helps reason about and compare different ETMs. An ETM can be completely characterized by a list of axiomatic definitions. This specification, however, cannot readily be used by a DBMS to *implement* an ETM for an application. Inspired by ACTA, Asset [33] allows users to define custom transaction semantics for specific applications. It provides transaction primitives that can be composed together to define a variety of ETMs. Asset still needs some higher layer, however, to appropriately organize its primitives based upon the available semantic information.

The Transaction Specification and Management Environment (TSME) [54] is closest to our approach. TSME provides a transaction specification language and a programmable transaction management mechanism (TMM) that configures a run-time environment to support a specified ETM. TMM translates a transaction model specification into a set of instructions and assembles run-time support from a transaction processing toolkit. One drawback is that all the components of the resultant system appear to be built from scratch, and there seems to be no way to integrate a TMM with an existing DBMS.

Barga and Pu have designed a Reflective Transaction Framework to implement extended transaction models [12]. Using transaction adapters, add-on modules that are built on top of an existing TM, they show how to extend the underlying functionality of the Encina [225] transaction processing monitor by capitalizing on the callback functionality provided by Encina. This is very similar to our approach at utilizing the mediator architecture of our TM component. The primary difference with our work is that we have designed the CORD language for specifying the extensions to TM, while they follow a programming approach. It should be possible to integrate our engine with their framework.

In lock-based TMs, the most common means of extension is to provide additional lock modes, or allow new ones to be defined. Most lock-based systems use a matrix to record lock compatibility information (e.g., Exodus and ObServer [66]). Modifying this information would be difficult in systems where there is either no defined "matrix" of locks (e.g., the logic for compatible locks is spread throughout the system), or the defined matrix is not meant to be altered (e.g., the matrix is stored in a C header file and lock modes are pre-defined constants). If new lock modes can be added to a matrix table, the core functionality of the system will be affected when new lock modes are requested. To use these new lock modes, however, the application designer might have to modify and rebuild parts of the system.

Some DBMSs provide support for defining new lock modes as needed, without any recompilation. The TM in the MARVEL process-centered environment [31], for example, determines lock modes from a fully-configurable lock matrix file; each MARVEL task encodes the lock modes it will request from the TM. A configurable lock-matrix, however, is not powerful enough to provide fine-grained control; for example, **AL** could not be implemented solely by a complex matrix. The TM could always be modified to acquire semantic information when determining lock conflicts. Barghouti [13] designed a TM that used a special-purpose language for programming concurrency control policies for rule-based software development environments (RBDEs). His TM extracted seven pieces of semantic information from RBDEs and had a language for specifying concurrency control policies. This approach was hard-wired since the TM directly inspected data structures from the RBDE and the language was specially designed for RBDE. Our work generalizes and extends Barghouti's ideas for wider applicability.

An alternative to serializability as a correctness model is the *checkout model.* In checkout, transactions operate on private copies of data that are checked out from a repository. The only contention for shared objects occurs when a transaction checks in/out an object. We view checkout, versions, and configurations as the domain of the application rather than something to be imposed by the TM. See [118] for a survey of extended checkout models.

11.8 Contributions and Future Work

Advanced database applications use databases to store information but they require more sophisticated concurrency control policies than standard DBMSs provide. Fortunately, such applications contain semantic information that describes their transactional needs. The transaction manager needs to incorporate such semantic information to provide the appropriate services to these advanced database applications. Our main contributions are:

- A mediator architecture that allows generic extraction of semantic information from an advanced database application to support an extended transaction model.
- The CORD language, a sample Concurrency Control Language that specifies the extensions to serializability needed for an extended transaction model.
- A CORD run-time engine that incorporates the semantic information to extend the transaction manager for a DBMS. The CORD engine uses the semantic information extracted from the application to match concurrency control policies in a CORD specification.
- Successful application of the CORD approach to implementing **AL** within Exodus.

For future work, we plan on carrying out more experiments with CORD and existing DBMSs. Once a particular set of CORD rules becomes fixed for an ETM, the run-time support would be more efficient if the rules could be compiled into native code, thus avoiding the cost of interpretation; we are currently investigating such an approach. We have focused our attentions on the concurrency control aspects of ETMs, but have not discussed the interaction and relationship that concurrency control has on recovery. In the same way that CORD rules can tailor concurrent behavior, it seems likely that a similar language-based approach can be used to program the recovery of ETMs.

Chapter 12

Low-Bandwidth Operation

Abstract

Software Development Environments have traditionally relied upon a central project database and file repository, accessible to a programmer's workstation via a high speed local area network. The tele-commuting paradigm has demonstrated the need for a new model, which allows for variable bandwidth machines to assist programmers in their development. A new client-server model is introduced which minimizes network traffic when bandwidth is limited. This research enables programmers to continue working within the framework of a Software Development Environment, without continuous high speed network access.

12.1 Introduction

A multi-user software development environment (SDE) supports collaboration among multiple participants in large-scale software engineering projects. It provides a repository in which source code, object code, documentation, test cases, etc. reside, with some form of concurrency control to coordinate access to shared files. It integrates a collection of tools, ranging from editors and compilers to configuration managers and modification request systems, and generally tracks the progress of the project. A subclass of SDEs, called process-centered environments (PCEs), in addition provide some formalism through which a *process* may be specified — basically a partial ordering among software engineering tasks, constraints and obligations of those tasks, and the files and tools used in the tasks [113]. The generic PCE kernel is parameterized by the desired process written by the process architect, and the same PCE can support a wide range of different processes for different projects.

Oz is a decentralized process centered software development environment in which the process is defined by a set of condition/activity/effects rules [27]. Oz is constructed around an object oriented database (objectbase) [144] which maintains persistent information about a process state. Oz objects map to a file repository, and each object corresponds to a directory containing file attributes. An instance of Oz represents its process internally by a rule network, whose links indicate possible forward and backward chains between rules related by a common predicate [108]. When a user requests to execute a particular software engineering task, Oz employs the network to enforce and automate the subprocess involving the rule corresponding to that task. If the rule's condition — a complex logical clause — is not already satisfied in the objectbase, backward chaining attempts to execute other rules, one of whose effects might satisfy the original rule's condition. Its activity, usually the invocation of an external tool, cannot be initiated until the condition is true. After an activity completes, one of the rule's effects — each a sequence of predicates — is asserted, and forward chaining triggers any other rules whose conditions have now been met. There are multiple disjoint effects to reflect the multiple possible results of a tool invocation (i.e., various success and failure cases).

Each participant in a process interfaces to the system through a separate client, which supplies the user interface. Traditionally, the Oz client is also responsible for invocation of external tools. The clients are coordinated by a server that incorporates the process engine, objectbase and the shared file repository [31].

The Oz server is only active when a client is logged-in to an environment. When the server is running, it writes the TCP port number it is bound to and the name of the host on which it is running, into a file in the environment directory called `.server_port`. When a client needs to connect to a server, it searches the environment directory for the `.server_port` file in order to extract the hostname where the server is running, and TCP port number it is listening to. This information enables the client to establish a connection to the server. In the event that no `.server_port` file is found, the client contacts the Oz daemon which is always active and bound to a well known port. The daemon is responsible for starting a server in the environment directory corresponding to a path passed into the daemon from the client. Once the daemon has successfully started the new server, it returns the new TCP port number which the server is listening on, to the client.

The standard client/server protocol is for the client to display the repository (objectbase) in graphical format, and the user selects from a task menu and provides the desired arguments. The client then transmits this information to the server for any needed backward chaining. To execute an activity, the server submits the tool invocation information back to the client, and then goes on to accept the next message from its input queue.

One of the key aspects of Oz is its decentralized structure. In order to support collaboration amongst geographically dispersed teams of users, Oz allows the modeling of Summits - the coordination of rules and associated tool invocations using data from multiple Oz sites, where each site corresponds to one environment directory with at most one active server. Each Oz site can function autonomously, but can also sign a Treaty with other sites in order to facilitate intra-site coordination. Each Oz client has a "local" process server that it connects to when started. Through the client interface, it is possible to establish connections to "remote" servers - a required step prior to running a Summit rule. If a user does not need to access data from a "remote" site, the client need only establish a connection to its "local" server.

In order to integrate unmodified tools into an Oz environment, a generic wrapper called a shell envelope is used. Shell envelopes resemble UNIX shell scripts, but are capable of both receiving input parameters from an Oz client and sending a return value back, indicating the success or failure of an envelope. The client executes a shell envelope whenever a software task requiring tool invocation is performed. After the shell envelope finishes, the client sends the return code to the server. Based upon the return value, the server chooses which effect to assert, which eventually carries out any consequent forward chaining.

Oz and other similar PCEs work well in an office or lab environment where workstations are connected via a high speed network. The networking capabilities of these machines are used to facilitate communication between PCE clients and servers. A file repository containing all project related files is maintained by a server and is accessed by the clients via the network. Due to these constraints, PCE designers have virtually ignored the possibility of work without a high speed network, whose very existence was an underlying assumption of the PCE design. LAPUTA is an extension to Oz which considers the full spectrum of network connectivity, in order to support PCE usage in non-traditional work environments.

12.2 Motivation

Software engineers are well-known for their long working hours, some of which can be conducted at home using dumb terminals and modems. This mode of operation is relatively easy for an SDE to support — *if* one does not mind giving up many of the advantages of modern workstations, notably the large graphics displays. A full-scale workstation could be installed at home, but conventional modem speeds make it infeasible to treat this workstation as just any other node on the network. Low bandwidth serial line protocols such as SLIP [197] and PPP [177] are inadequate for maintaining a sophisticated display, or transferring large files for local tool manipulation. X11 based protocols such as XRemote [46] and LBX [72] will maintain higher X11 throughput via a low bandwidth connection established between a pair of modems, but may still be too slow for interactive usage. To date, the SDE community has nearly ignored the possibility of off-site access, and the best that can be expected is a TTY user interface simulating the capabilities of the standard (graphics-based) user interface accessible only to users communicating over a local area network.

The proliferation of powerful home computers, and the recent increase in tele-commuting [167] point towards a demand for applications tailored to run on these new computers. One obvious difference between office machines and their home counterparts is the availability of network access. Most machines in offices and labs are connected to high speed local area networks. In comparison, home computers are typically connected to the network via modems at reduced communication speeds. Client/server applications used on home computers have to adjust to deal with increased network latency, and reduced throughput.

The tele-commuter thus introduced a new challenge for SDEs. Home PCs provide essentially the same compute power as their office counterparts, but with low, perhaps varying bandwidth. The challenge was to adapt the underlying PCE architecture that normally relied on (at least) a shared network file system and high speed network access to the PCE server from the client, to support more innovative work environments.

12.3 Goals

The goal of this research was to investigate the problems of low bandwidth PCE operation, and to develop a system which overcame those problems within the framework of Oz. To facilitate low bandwidth operation, a new client/server model is introduced which re-thinks the responsibilities of the client in order to minimize network traffic. A user is better able to exploit wireless network connections by minimizing communication, hence reducing power consumption. On "pay per packet" networks, the reduction in network bandwidth leads to direct cost savings for the end user, and users connected via slow modem connections can increase their productivity. Through the use of the LAPUTA client, a user is able to take advantage of all of the benefits of an SDE from any location, with even minimal network connectivity.

12.4 Sources of Network Traffic in Oz

In order to minimize network bandwidth requirements, some of the traditional responsibilities of the client are re-targeted. To reduce the communication between an Oz client and server, it is important to understand the traffic produced by the Oz client/server architecture. There are over 40 messages types sent between the various Oz clients and servers. Most of these message types can be characterized as one of the following:

Client to server message types:

- Login requests
- Rule firing requests
- Software tool responses
- Server queries
- Remote server status information
- File transfers

Server to client message types:

- Login responses
- Objectbase updates
- Query responses
- Tool invocation requests
- File transfers

In addition to the network traffic generated by the client/server protocol in Oz, it is imperative to consider the NFS traffic produced by a client executing a tool which requires accessing a file stored in the server's repository.[1]

12.4.1 Client Login

When an Oz client connects to its "local" server, it must first transmit some login information. In particular, the client must describe its interface type, the mode of operation (standard user or administrator), and the name of the user running the client. This login message is fairly small, and is typically under 60 bytes long. A client login message sent to a "remote" server additionally sends information about other servers, which the client is already attached to. This information which includes the server's hostname, IP address and TCP login port is used by the remote server to establish communication with other servers during a Summit. With this added information, the length of the login message increases, but still remains relatively small.

In contrast to the short client login messages in Oz, the server's response can be very large, and varies widely in size across different environments. The server first sends the client a server ID which is unique amongst all of the servers collaborating in a multi-site Oz environment. The server then sends a unique client ID which the client uses to tag all future message to the server for identification purposes. Following the 2 IDs, the server sends all of the objectbase information required for the client to create a graphical representation of the server's objectbase. The "local" server also sends the client a class table, rule table, and rule network.

[1]Oz was built upon NFS, so files from the server's repository may actually be NFS served by a host other than the machine running the Oz server.

12.4.2 Objectbase Image

Oz objectbases can become quite large, populated with many objects over the course of a project lifetime. It is not unusual for an objectbase to have thousands of objects. One of the environments used for developing Oz[2] contains 972 objects as of this writing. In order for the client to draw an objectbase image, it only needs to be sent information about the objectbase structure, not the attribute values for the objects. For each object in the objectbase, the server packs a buffer containing the object's:

- name
- unique ID
- class name
- version number
- list of children
- list of links

Root objects (the objectbase is a forest structure with multiple roots) are sent separately from the rest of the objectbase, as a means of identifying the root objects to the client. The client unpacks the objectbase and the list of root objects sent from the server into a local hash table, hashing the objects by their object ID. The client starts with the root objects and constructs a tree for each root object. Only those objects which are descendants of root objects are displayed in the clients objectbase image.

12.4.3 "Strategy" Data

A "local" server sends the client a class table, rule table, and a rule network, which are collectively known as the "strategy". The class table describes the objectbase schema for a particular environment, including the class names, their attributes, and class inheritance. The rule table lists all of the rules which can be fired by a user from the client. The rule network contains the rule signatures, and shows forward and backward chains between related rules. For complex environments with many rules and classes, the "strategy" data sent from the "local" server to the client during login can be in the tens of kilobytes.[3]

12.4.4 File Access

The greatest amount of network bandwidth consumed in the Oz client/server model is caused by the NFS traffic produced when a client executes an activity which uses files from the server's repository. Even with the most sophisticated modems and compression techniques, transferring a large software source file over a telephone line for editing can take prohibitively long. A matched pair of 14,400 baud modems can only transfer about 1KB of data per second[4]. Attempting to do more complex tasks such as compiling a file are virtually impossible due to the large number of input and output files (header files, object code, executables) that are processed.

Aside from the network traffic generated by accessing files across the network in order to run software tools, the only significant source of traffic is data sent to the client during login to the server(s). In designing a Low Bandwidth client model that operates within the framework of Oz, it is therefore critical to solve the file transfer problem, as well as reduce the bandwidth consumed during login.

[2]The development of Oz is now carried out within a production Oz environmental.

[3]The strategy data associated with the oz_master environment used to develop Oz is 10,730 bytes long.

[4]Actual throughput varies depending on which error correction protocol is used, transport protocol employed, and the quality of the phone line.

12.5 Low Bandwidth Client

The goal of the Low Bandwidth client in Oz was to present the user with a consistent client interface to the SDE, without requiring high speed network access, or sacrificing functionality. Through intelligent caching, almost all of the network traffic caused by a client logging into a server can be eliminated. Prior to LAPUTA, some work had already been done to reduce the network traffic. Specifically, the update information that the server sends to a client regarding changes made to the objectbase was minimized. Rather than sending a complete image of an Oz objectbase to a client every time a structural change is made to the objectbase, the server sends the client a `diff` with respect to the previous version of the objectbase. The server maintains a queue of objectbase changes, and periodically flushes the queue, sending the updates to the client.

12.5.1 Objectbase Caching

In the LAPUTA Low Bandwidth client, all objectbase display information is read and written to a cache file contained within the client's environment cache directory. The cache contains a list of all of the objects in the objectbase, their ID, name, class name, a list of children objects, a list of objects which are linked-to and a version number.[5]

The server plays an active role in supporting the Low Bandwidth client's objectbase caching scheme. Each object in the server's objectbase contains a persistent version number, and no two objects in the objectbase share the same version number.[6] Version numbers are 32 bit unsigned integers, and are allocated by the server in a monotonically increasing fashion. The server keeps track of the highest version number already in use, and always adds 1 to that value prior to updating an object's version number.

The object version numbers are used to identify structural changes to the objectbase. As such, all newly created objects get allocated a version number which is `MAX_CURRENT_VERSION + 1`.[7] Furthermore, any object which is involved in a structural change to the objectbase in any way has its version number updated to the value `MAX_CURRENT_VERSION + 1`. In the case of an `Add` operation, the new object as well as the parent object of the newly created object are assigned new version numbers. Similarly, if an object is deleted from the objectbase, the parent of the deleted object has its version number updated.

When a client closes a connection to a server, it writes out all of the current objectbase information it has to an objectbase cache file. A separate objectbase cache file is maintained for each site in a multi-site environment. When a client exits normally, it writes out an objectbase cache for each site that it was still connected to prior to shutdown. The name of the objectbase file is `objectbase_cache.local` for the objectbase corresponding to the "local" server's objectbase. All other objectbase caches are named `objectbase_cache.XX` where `XX` is the remote server's unique server ID. For ease of processing and machine byte code interoperability (big endian vs. little endian), the objectbase cache files are written in ascii format.

When a Low Bandwidth client is started up, it searches for the `objectbase_cache.local` file in its environment cache directory. If this file is found, it is read in and processed. Each object which is read from the cache file is inserted into an object hash table that the client maintains. When a Low Bandwidth client connects to a "remote" site, it searches for the objectbase cache file corresponding to that site, and inserts additional objects into the hash table. Each object in the hash table is marked with the ID of the environment which it belongs to, so that the client may distinguish between objects from different sites. As the client reads in the objectbase cache and populates the object hash table, it searches for the highest object version number, among all of the objects being processed.

[5]In the database community, a version number could also be called a generation number.

[6]Version number 0 is a special initialization value which may apply to multiple objects in the objectbase.

[7]The Oz implementation uses a global variable defined in the Darkover object management system called `GlobalVersionCounter`.

Part of the Low Bandwidth client's login protocol includes the transfer of the largest version number the client extracts from its objectbase cache. When the server processes the Low Bandwidth client's login message, it unpacks the version number and compares it against all of the objects contained within its objectbase. All objects which have version numbers which are larger than the version number passed in by the Low Bandwidth client are packed into a buffer, and sent to the Low Bandwidth client as objectbase cache updates. When the Low Bandwidth client receives the objectbase updates, it unpacks all of the objects in the incoming buffer, and inserts them into the object hash table. If an object is inserted into the hash table and another object with a matching server ID and object ID is found in the hash table, the new object replaces the old object.

Using the objectbase cache algorithm outlined above, it is possible for spurious objects to be left in the Low Bandwidth client's object hash table temporarily. This can happen if an object is removed from the server's objectbase while a Low Bandwidth client is not connected to the server. The server does not notify the client of which objects have been deleted from the objectbase, but instead sends objectbase updates for the parent objects of the defunct objects. This makes it possible for the client to have in its hash table a non-root object without a parent. This situation does not however cause problems, as the client is aware of what its top level objects truly are (the server always sends the client a list of top level root objects), and the display algorithm for drawing the objectbase always starts at the root objects in the objectbase, and only draws those objects which are descendants of an objectbase root. Similarly, the routines used to write out the Low Bandwidth client's objectbase cache will only write out those objects which are descendants of the objectbase roots. This prunes out spurious objects from the object hash table in a lazy fashion.

We have found that typical Oz objectbases in production environments supporting software development have fairly static objectbase structures. It is therefore possible to greatly reduce the network traffic associated with the server sending a large objectbase to the client, and only send a minimal set of objectbase updates to the Low Bandwidth client. In an environment with a relatively large objectbase, this can have a dramatic effect, significantly reducing the network traffic a server sends to a Low Bandwidth client at login.

12.5.2 Strategy Caching

While Oz provides powerful mechanisms for evolving an environment's process as well as its objectbase schema, these tools are seldomly used once an environment reaches a mature state. Since the "strategy" data that an Oz server sends to its "local" clients can be arbitrarily large, and changes infrequently, the "strategy" data is an obvious candidate for caching by the LAPUTA Low Bandwidth client.

When an administrator modifies and environment's rule table, rule network or schema via the Oz `evolver` or `loader`, a persistent value stored in an environment filed called `.strategy_update_ctr` is incremented. This value is intended for use by the server for treaty validation, however it is reused for the purposes of caching the "strategy" data in the Low Bandwidth client's environment cache directory.

When a Low Bandwidth client is started, it searches for the `strategy_cache` file in its environment cache directory. If this file is found, the first 4 bytes of the file, which contain the value corresponding to the `.strategy_update_ctr` are read in. This value is sent to the "local" server during the Low Bandwidth client login. The server extracts the `.strategy_update_ctr` value from the login message that the Low Bandwidth client sends to the server. This value is compared against the value in the server's current `.strategy_update_ctr` file. If the values differ, the Low Bandwidth client's "strategy" cache is rebuilt. The server sends a response to the client's login message, which includes the new value from the `.strategy_update_ctr` file, as well as the actual "strategy" data. If the server does not send any new "strategy" data to the Low Bandwidth client, the client assumes that the "strategy" data contained within its cache file is valid. Any time the client receives "strategy" data from the server, it writes it into a its local `strategy_cache` file.

12.6 Running External Tools

In Oz, the server submits tool invocation requests to the client for local processing. When the client receives a tool invocation request message[8], the message gets passed to the client's Activity Manager, which is the client component responsible for running external tools.

In order to perform black-box integration of unmodified tools in an Oz environment, the Activity Manager runs a wrapper called a shell envelope, which is a shell script capable of receiving input parameters from the client's Activity Manager, and passing envelope return values back to the client. The server constructs a buffer containing the name of the shell envelope the client must run, as well as the parameters that are to be passed into the shell envelope.

The Activity Manager creates input and output pipes for communicating with the envelope, via the `pipe()` system call. For handling the return value from an envelope, a named pipe is employed instead of the Unix pipes used for the envelope input and output. The use of the named pipe allows the client's Activity Manager to read envelope return values, even after the envelope process has died. Once all of the input, output and return value pipes have been setup, the client calls `fork()`. The child process which is created by `fork()` calls `exec()` to run the shell envelope. The input and output pipes are created such that the envelope's `STDIN` file descriptor is the reading end of the output pipe created by the Activity Manager, and the `STDOUT` and `STDERR` file descriptors are the writing end of the input pipe created by the Activity Manager.

The client creates a graphical interface to the envelope called an `RFRAME`. Any data read on the client's input pipe is output of the shell envelope, and is displayed in the `RFRAME`. Any data which the user types into the `RFRAME` is sent to the envelope via the writing side of the output pipe. In this way, the user can read and write data to a tool through the `RFRAME` graphical interface provided by the client. The `RFRAME` also has buttons which the user can "click" on in order to send `SIGTSTP` and `SIGCONT` signals to the envelope process in order to pause and resume envelope execution respectively.

Once a shell envelope has finished running, the client which is the parent process of the envelope is sent a `SIGCHLD` signal from the operating system. Upon receipt of the `SIGCHLD`, a signal handler previously registered by the client is called. The signal handler reads the return values sent by the shell envelope to the client, via the named pipe. This value is sent back to the server for post processing.

12.7 Proxy Client

In the context of a multi-user SDE operating over a high speed local area network, the placement of the Activity Manager into the Oz client was appropriate. By not burdening the server with activity invocation, as well as the added load a machine incurs from performing an activity, the server was able to operate more rapidly [31]. By distributing its computing load to client hosts, the overall power of the system was increased. Furthermore, by placing the Activity Manager in the client, it is possible to create multi-platform environments[9], and tools can be invoked with the user ID and associated permissions of the actual user, rather than the user ID of the server.

When attempting to operate an Oz client over a low speed network connection, the current location of the Activity Manager is a great hindrance to performance. In LAPUTA, the Low Bandwidth client's Activity Manger is assisted by a new class of client. Rather than having the Oz server send an activity invocation message to the Low Bandwidth client, which would then produce NFS traffic to supply needed files to the client[10], a new Activity Manager is created that runs on a host local to the server. Local in this context implies a high bandwidth network connection to the host machine executing the server, not necessarily the same host. A new client called a Proxy client is introduced, which contains the new Activity Manager, but unlike other clients does not act as a user interface to the system.

[8]The actual message type is `EXECUTE_MSG` in Oz.

[9]Oz clients currently exist for SunOS and Windows NT

[10]Assuming the client had the ability to NFS mount the filesystem where the files reside.

12.7.1 Proxy Login to "Local" Server

A Proxy client is started manually by the user of the Low Bandwidth client, after the Low Bandwidth client has already finished its login procedure. Since the server must be running in order for the Low Bandwidth client to be running, and since the Proxy client is started after the Low Bandwidth client, a server is always running prior to the Proxy client. The Proxy client therefore never needs to worry about contacting the Oz daemon in order to start the server. The Proxy client simply opens the server's `.server_port file` and extracts the hostname and TCP login port number of the currently running server. The Proxy client then connects to the server by sending a `PROXY_LOGIN_MSG` message. An optional command line argument to the Proxy client allows the user to specify the Low Bandwidth client ID which the Proxy client is being started for. The Low Bandwidth client ID if specified is sent with the login message to the server.

12.7.2 Proxy to Low Bandwidth Binding

Each Proxy client runs on behalf of one Low Bandwidth client[11] If the Proxy client sends a specific Low Bandwidth client ID to the server with its login message, the server attempts to bind the Proxy client to the specified Low Bandwidth client. If no client with the specified Low Bandwidth client ID is not found, the Proxy client's login connection is closed and the Proxy client exits with an informative message for the end user. If no Low Bandwidth client ID was specified, the server attempts to match the Proxy client with the correct Low Bandwidth client. The server finds all Low Bandwidth clients logged in which do not yet have Proxy clients bound to them. For each Low Bandwidth client without a bound Proxy client, the server determines if the Low Bandwidth client is running with the same user name as the user name associated with the new Proxy client. If the user names match, the Proxy client is bound to the Low Bandwidth client in the server's `ClientDescriptorTable`. If no Low Bandwidth client is found for the Proxy client, the Proxy client's connection to the server is closed, and the Proxy client notifies the user that no appropriate Low Bandwidth client was found and then exits.

Unlike most other proxy models, the LAPUTA Proxy client is not an intermediary that sits between a client and a server. Instead, the Proxy client is a peer of the Low Bandwidth client. Both the Proxy client and the Low Bandwidth client establish independent communication channels with the Oz server. Rather than burden the server with exchanging information between the Proxy and Low Bandwidth clients, the proxy model in LAPUTA establishes a TCP connection between the Proxy client, and the Low Bandwidth client the server has bound it to. When the Low Bandwidth client first logs into the Oz server, it creates a TCP login socket, and registers a callback function on the login socket. The TCP port number of the login socket is sent with the Low Bandwidth client's login message, and is stored by the server in the `ClientDescriptorTable`.

Once the server determines which Low Bandwidth client to bind a Proxy client to, the reply message to the Proxy client's login contains the hostname and TCP login port of the Low Bandwidth client. This information is used by the Proxy client to establish a connection with the Low Bandwidth client, closing the triangle of communication between the Low Bandwidth client, Proxy client, and the "local" server. When the Proxy client connects to the Low Bandwidth client, the callback function registered by the Low Bandwidth client is executed[12]. This function calls `accept()` to accept the connection from the Proxy client, and sends connection information about any "remote" sites the Low Bandwidth client is attached to, to the Proxy client. The Proxy client receives the information about the "remote" sites, and attempts to establish connections to the remote servers, passing in the Low Bandwidth client's ID with the "remote" login message to facilitate proper Proxy to Low Bandwidth client binding in the "remote" servers.

[11]In the related Rivendell project, a Proxy client can service multiple Oz clients.

[12]The function is called `x_handle_proxy_login`

12.7.3 Proxy Execution

Once a Proxy client has established a connection with the server as well as the Low Bandwidth client, the server is able to take advantage of the Proxy client. By default, any tool invocation message (`EXECUTE_MSG`) that the server would have sent to the Low Bandwidth client, is redirected to the Proxy client. The server also sends a message to the Low Bandwidth client of type `PROXY_EXECUTE_MSG`, to alert the Low Bandwidth client that there is an activity being run on its behalf, by the Proxy client, and the Low Bandwidth client creates an `RFRAME` interface for the activity. The `EXECUTE_MSG` sent to the Proxy client and the `PROXY_EXECUTE_MSG` sent to the Low Bandwidth client contain a common identifier called **`tm_data`**[13]. The `tm_data` value is used by all three parties (Proxy client, Low Bandwidth client, and server) to tag messages about an activity, as pertaining to a specific rule chain. The Low Bandwidth client associates a single `RFRAME` with each `tm_data` value.

Upon receipt of an `EXECUTE_MSG`, the Proxy client passes the activity invocation request to its Activity Manager. The Proxy client's Activity Manager is functionally very similar to the standard Oz client's Activity Manager. The one notable difference is in the treatment of envelope input and output. Unlike the traditional Oz clients which provide user interfaces, the Proxy client does not provide any user interface of its own, and relies upon the Low Bandwidth client for user interaction. When a Proxy client's Activity Manager reads envelope output data on its input pipe, the data must be displayed in the Low Bandwidth client's `RFRAME`. The Proxy client determines the `tm_data` value associated with the child process which created the output data. The `tm_data` value and the output data are packed into a buffer, and sent to the Low Bandwidth client with a message type `PROXY_TO_RFRAME_MSG`. When the Low Bandwidth client receives the `PROXY_TO_RFRAME_MSG`, it determines the `RFRAME` associated with the `tm_data` value sent by the Proxy client, and writes the output data to the appropriate region of the `RFRAME`. If the user enters input in an `RFRAME` which is associated with an activity being performed by the Proxy client, the data is packed into a buffer along with the `tm_data` value for the `RFRAME` and sent to the Proxy client with message type `RFRAME_TO_PROXY_MSG`. The Proxy client upon receipt of the `RFRAME_TO_PROXY_MSG` determines the output pipe associated with the `tm_data` value passed in by the Low Bandwidth client, and writes the data to the desired child process via the output pipe. By sending tool input and output messages between the Low Bandwidth and Proxy clients, a tool is able to execute on the Proxy client but from the end users perspective appears to be running on the Low Bandwidth client.

When the Proxy client's Activity Manager determines that a child process has died, it reads the envelopes return value through a named pipe. This return value is sent back to the server for processing along with the activities' `tm_data` value, with a `DONE_MSG` message. When the server receives a `DONE_MSG` message from a Proxy client, it processes the message as if it had come from the Low Bandwidth client. The server is thus able to process the envelope return values from the Proxy client, using the same code developed to process a traditional client's return value. This reduced the complexity of the changes made to the Oz server[14].

12.7.4 Support for Graphical Tools

To support non-interactive text based tools such as a compiler, the LAPUTA Proxy client model works very well. Such tools are invoked by the Proxy client, and all output messages appear on the screen of the Low Bandwidth client. It is important to note however that some tools such as editors are interactive, and often have graphical user interfaces. To run these tools correctly, the Proxy client sets its `DISPLAY` variable[15] to point to the host on which the Low Bandwidth client is running. Once this step is taken, any graphic based tools which are run by the Proxy client, displays its graphical output and read its input from the host running the Low Bandwidth client.

[13]The name `tm_data` is an artifact of legacy code in Oz.

[14]Fewer than 2,000 lines of server code were changed in order to add support for the Proxy client in the Oz server

[15]X11 based tools reference the environment shell variable `DISPLAY` to determine on which display server a client is drawn

This remote usage of interactive tools creates previously non-existent network traffic, by sending X11 packets over the network connecting the Low Bandwidth client to the Proxy client.

12.8 Low Bandwidth Activity Manager

For tools which produce alot of X11 based network traffic, it is sometimes counter-productive to run the tool from the Proxy client. From a bandwidth utilization standpoint, it is only sensible to run an external tool via the Proxy client's Activity Manager if it results in an overall reduction in network traffic. If a graphical based tool generates more X11 network traffic than would be required to transfer the files needed to run the tool to the Low Bandwidth client, the tool is better suited to run locally on the Low Bandwidth client. An environment can be constructed in such a way that a certain subset of tools are run by the Low Bandwidth client's Activity Manager, or such that the end user can determine where the tool will run.

The Low Bandwidth client contains a modified Activity Manager. Since we cannot assume a Low Bandwidth client has NFS access to a server's file repository, the Low Bandwidth client's Activity Manager, and the server have added support for transferring files to and from an Oz server. Prior to the server sending the Low Bandwidth client an activity invocation message (of type `FILE_TRANSFER_ACT_MSG`), the server must make sure that all of the necessary files have been transferred to the Low Bandwidth client. The server saves state information about the activity, and then context switches. Only once all of the files needed to start the activity have been sent to the Low Bandwidth client does the server restore its context, and send the `FILE_TRANSFER_ACT_MSG` to the Low Bandwidth client to start the activity. When the server receives a return value from the Low Bandwidth client's Activity Manager, it must once again context switch. The return value is not processed until all of the files which were modified by the Low Bandwidth client have been copied back to the server's file repository. The LAPUTA implementation in Oz thus adds 2 context switching points into the Oz server.

12.8.1 File Transfers Support

In order to support the Low Bandwidth client's Activity Manager, the server must be able to both send and receive files to the Low Bandwidth client. Rather than use existing file transfer utilities such as `ftpd`, a new file transfer protocol was written and implemented within the Oz server and the Low Bandwidth client. Using an existing file transfer utility would have required the overhead of starting a new process, as well as creating and maintaining inter-process communication channels to the file transfer utility, from both the Oz server and the Low Bandwidth client.

Since a file transfer can take arbitrarily long based upon the size of the files being transferred and the speed of the network, the server must be careful to not block its ability to service other clients and servers. The server always maintains a list of files that need to be sent to a client in the `ClientFileExportList` and files that need to be sent from the client to the server in the `ClientFileImportList`. In the server's main IPC processing loop, in the absence of any entries in the `ClientFileImportList` and `ClientFileExportList` lists, the server issues a blocking `select()` call. Whenever the server receives a new IPC message on one of its connected sockets, or its login socket, the call to `select()` returns, waking up the server process. To support background file transfers, the server checks the file transfer lists in a function called `calc_file_counters()` to see if there are any files which are ready to be transferred on the import or export lists. If the server determines that there is a file to transfer, the server issues a non-blocking `select()` call. If `select()` returns a value of 0, indicating that there are no sockets with data waiting to be read, the server jumps into the file transfer code. In this fashion, the server only performs file transfers during periods of inactivity.

The file transfer routines find a target file to export or import from the export and import lists. Prior to sending the first block of a file being exported to a Low Bandwidth client, the server computes a checksum of the target file. The checksum is sent to the Low Bandwidth client and compared against any copy of the file that the client might already have in its cache. If the Low Bandwidth client finds the file in its cache and computes the same checksum as the value sent by the server, it notifies the server that the file need not be transferred. Once a server has determined that a file does in fact need to be transferred, it reads a 4K block of the file[16] and sends it to the Low Bandwidth client, with a message type `SEND_REMOTE_MED_ATT_MSG`[17]. The server determines a unique name for the file within the Low Bandwidth client's file cache directory (called `imported_files`), and always sends this filename along with file data. Once a 4K block of data has been written to the Low Bandwidth client, the server returns to the top of its IPC loop. To enact a simple flow control mechanism, the server waits to receive an acknowledgement message of type `RCV_REMOTE_MED_ATT_MSG` from the Low bandwidth client before transferring any additional file data to that particular client.

Importing a file from a Low Bandwidth client is very similar to exporting a file. The server determines which file needs to be imported, computes the local checksum of the file (if the file exists locally), and the name of the file within the Low Bandwidth client's file cache directory. This information is then sent to the Low Bandwidth client which computes its local file checksum. If the server and Low Bandwidth client's file checksums match, the client notifies the server, and the server removes the file from its `ClientFileImportList` list. If the two checksums do not match, the client sends back the first 4K block of the file. The server than requests additional 4K blocks of the file during its idle cycles until the entire file has been transferred.

12.8.2 Language Extensions for Low Bandwidth Client

The determination of whether a tool is more appropriate to run on the Proxy client or on the Low Bandwidth client is not made by the server. In LAPUTA, the process administrator is provided modeling facilities that can define a set of tools which should be executed by the Low Bandwidth client, rather than the default execution model of the Proxy client. For tools with relatively short source files compared to their X11 display requirement, such as `xdvi` and `ghostview`, the administrator would add a flag into the toolbase, defining a tool that would never run on the Proxy client. Such a definition might resemble the following:

```
LB_TOOLS :: superclass TOOL:
  [
    lb_execution_flag : true;
  ]

  display_ps    : string = ``display_ps PS_FILE.contents'';
  display_dvi   : string = ``display_dvi DVI_FILE.contents'';
```

The boolean value `lb_execution_flag` is set to `true` when a tool is to be run by the Low Bandwidth client, and defaults to false.

In certain circumstances, it is not possible to know in advance if a tool will perform better when run by the Proxy client, or by the Low Bandwidth client. A straight-forward solution is to model two different activities, one of which runs on the Low Bandwidth client and the other which runs on the Proxy client. The determination of which Activity Manager to use can then be made by the end user.

12.8.3 Low Bandwidth Summary

Low bandwidth operation in Oz is achieved through a combination of intelligent caching, and the delegation of activities to the Proxy client. These same principles could easily be applied to PCEs

[16]4K block size is chosen to prevent the TCP buffers from overflowing, and for filesystem `read()` efficiency.
[17]The term medium attribute in Oz refers to a file attribute.

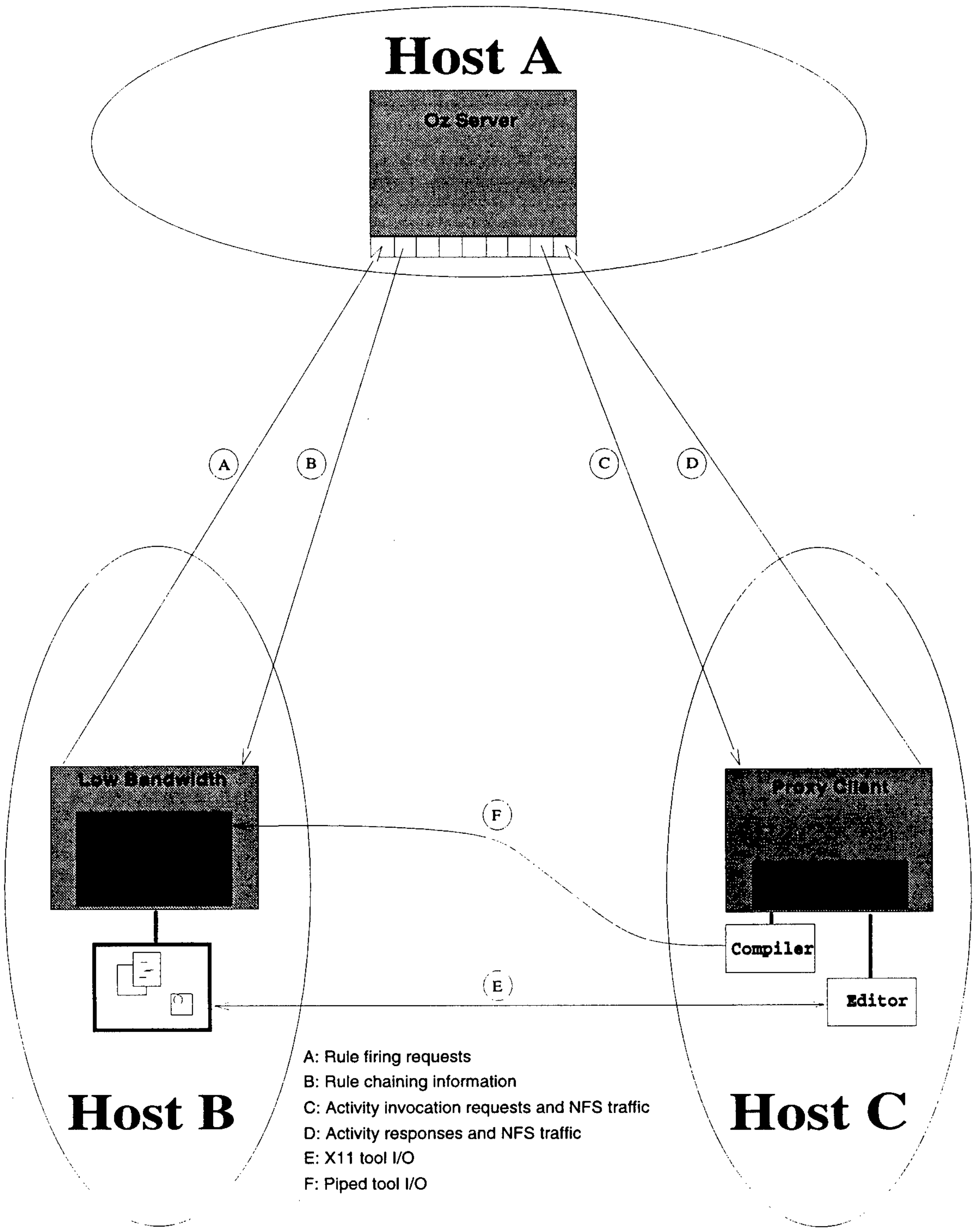


Figure 12.1: Summary of Low Bandwidth Client Network Traffic

and SDEs other than Oz. A summary of the network traffic that the LAPUTA low bandwidth model creates is shown in figure 12.1. In this figure, we envision an Oz server running on computer A. The user in this case is located at computer B. A Proxy client has been started on computer C. It is assumed that computers A and C share a high speed network, but that computer B is connected to the network via a low bandwidth connection.

There is another possible approach to low bandwidth operation. It is possible to run an Oz client on a machine which shares a high speed network connection with the "local" server, and which transmits the X11 based protocol generated by the user interface portion of the client to a remote display. This approach shares the same problems of interactive tool I/O which the Low Bandwidth client model addresses through file transfers and locally run activities. The software tools invoked by the client would run on the remote machine, and send all of their display information over the low bandwidth connection. In addition to the network traffic generated by the tools X11 output, the client's GUI would generate a significant amount of X11 based network traffic. Starting the Oz XView client in the oz_master environment (the environment used to develop Oz) would consume a total of 82,084 bytes of bandwidth. Starting the Oz Motif client in the same environment would use 190,172 bytes worth of bandwidth. With such high bandwidth requirements for graphical clients, sending a client's GUI display information across the low bandwidth connection is impractical.

12.9 Future Work

Even with an efficient low bandwidth client, we would often like to work on a project in a completely disconnected mode. This situation may arise when we do not have access to facilities that would allow us to establish a network connection from our current location. Even if it is always possible to establish a high speed network link (perhaps via wireless computing), it may be desirable to work in a disconnected mode, for economic or security reasons. Although wireless computing promises to give mobile computers network access regardless of location, this service will not come for free. Many companies also "firewall" their networks as a security measure, making it impossible to connect to the network [191]. In these companies, disconnected operation is the only viable option for employees who are not on-site.

Disconnected operation in this context is not viewed as a failure case. An alternative client could manage the explicit disconnection from the network and disconnected development. The disconnected client could be a traditional client augmented with a single user process engine. Prior to disconnecting from the network, the client would create a complete copy of the objectbase from the server and store a local copy of it on the target host. A subset of the files in the server's repository would also be copied onto the host. Once all of the data and file pre-fetching complete, a user would be able to remove a computer from the network and use local copies of the objectbase and files. After a network connection was reestablished, files modified by a disconnected user would be reintegrated into the server's repository.

12.9.1 Pre-fetching

Disconnected operation could be achieved through a combination of intelligent data and file pre-fetching in conjunction with a single user Oz process engine which replaces the functionality of a Oz server. In order for pre-fetching to be effective in LAPUTA, we have formulated a list of requirements that the system must adhere to:

- Be able to pre-fetch a working subset of files such that the user may be able to do development work locally.
- The fact that files have been copied to a disconnected LAPUTA client should not hinder the work of *other* users.
- Inconsistencies between local copies of files and those in the central repository must be easily trackable upon reconnection.

Data pre-fetching is straightforward. A complete objectbase image is copied from the server. A user may attempt to conduct work while disconnected which requires accessing files which have not been pre-fetched. A complete copy of the objectbase is required in order to determine which files if any are missing from the local repository. While it is possible to prune some attributes from the objectbase in order to only maintain critical information about each object, the space savings from such an action would be minimal, as objectbases are relatively small. Experience using Oz to develop large software systems has shown that a typical objectbase consumes less that .5% of the total file system space of a project, and grows proportionally to the project's overall size.

Unlike data pre-fetching, it is usually not possible to pre-fetch an entire file repository onto a mobile machine due to space limitations. I propose three possible approaches to the selection of files to pre-fetch, the last of which is novel:

- *Manual*: A user supplies an explicit list of objects whose files should be pre-fetched.
- *Heuristic*: The system maintains statistics about each user's past efforts, and assumes the same files are needed for future work. Statistics which might be kept include each individual user's recent file accesses, as well as which tasks each user has recently performed.
- *Process-based*: A user supplies an explicit list of tasks to be carried out while disconnected, and the system analyzes the process definition to determine which files are required for those tasks, as well as all tasks which the system may initiate due to the rule processors chaining mechanism.

An entirely *manual* approach puts the full burden of selection on the user. If a critical item is discovered to be missing after a the network connection has been broken, local development may be stalled. While it is assumed a user will be able to identify at a high level what type of work is desired to be accomplished during a planned period of dis-connectivity, a user may not always correctly identify all needed support files. An example would be a software engineer who pre-fetched some "C" source object's file to edit, but neglected to pre-fetch all of the header files required to recompile the source files. In this case, recompilation of the pre-fetched "C" file would be impossible, and the development would be limited to editing the file, but not recompilation. In a large project, it would be easy for some needed files to be forgotten.

Pure *heuristic* selection assumes inertia on the part of the user. That is to say, the system generally prepares for a user to continue doing essentially the same work while disconnected that was recently being performed while connected. Each user's recent development efforts would be tracked. It is assumed that any file which was accessed during recent development, would also be needed for future efforts. The system would keep track of which files were most recently accessed by each user. There is a tradeoff here between biasing the heuristics towards pre-fetching too little versus too much. If the user does not plan to repeat exactly the same task (which may have already been finished), the materials available may not be sufficient for the new work. Yet on a portable machine with limited disk space, we would like to prune out all unnecessary files from the local disk in order to preserve the precious commodity.

Process-based selection addresses the problems encountered with *manual* and *heuristic* selection. However, the process-based approach is not as simple as it sounds. Practical industrial-scale processes are complex, with numerous opportunities for choice or iteration [126]. Since each rule may have conditions which lead to backward chaining and effects which lead to forward chaining, we can construct a graph of a portion of the process, showing the relationships amongst specific rules. Expanding the graph to show the transitive closure of consequences emanating from a single rule can lead to very large graphs, and instantiating each rule's task with the appropriate files could mark most of the repository for pre-fetching. Figure 12.2 shows a very simple and small transitive closure of a possible rule network.

We combine process knowledge with heuristics from previous access patterns in order to prune the branching paths, producing the subset of files most likely to be needed. The system would always pre-fetch files required to fulfill any tasks which are reached through backward chaining, before those needed to support forward chaining of a user identified task. The ordering is meant to assure that the system pre-fetches all files required to perform the task which was originally identified by the user prior to disconnection. If disk space is limited, the system would make sure that all files required to fulfill backward chaining from a desired task are pre-fetched, at the possible expense of files which would be required for tasks reached through forward chaining. In this way, we could ensure that the user is able to at least perform the tasks which were identified by the user prior to disconnection.

In Oz, we maintain statistics on which of the multiple effects of a rule has been selected most frequently, with respect to this specific user and the arguments desired for the originating task[18]. If statistics are available for a specific task, the information could be used to restrict the expected forward chaining to a manageable level. By weighting the links connecting tasks in the rule network based on the statistical probability that a path will be followed, we can compute a probability graph. Once the rule graph is generated and the probability of transition-ing from each node of the graph is identified, it is possible to compute the likelihood of reaching each node in the graph. The probability graph of a sample rule network is shown in figure 12.3. If the system guesses incorrectly, and a rule which was identified as unlikely to be reached via forward chaining is in fact fired, the forward chaining must be delayed until reconnection when the needed files are accessible from the server's repository. The degree to which file selection is pruned can also be adjusted to accommodate the varying size of a local disk, e.g., by not completing even the most likely forward chaining path if the disk is too small and considering multiple paths if there is more free space.

The disconnected model would allows for a process architect to distinguish between `long duration` and `short duration` tasks. An example of a `long duration` task might be editing a file, where as a `short duration` task might be removing a file from a revision control system. The distinction between `long duration` and `short duration` tasks further helps prune which files get pre-fetched. Upon a request to disconnect, the process engine would follow the backward chains emanating from a desired rule firing, and attempt to execute any needed tasks to satisfy the original rule. If a rule's task is of type `short duration`, then the actual time consumed by executing the task is small, and it is done prior to disconnection. This differentiation is important as it would allow us to do necessary work prior to disconnection, hence reducing the number of files that need to be pre-fetched in order to satisfy backward chaining. The process engine would stop firing rules as soon as a `long duration` task (such as editing) were encountered.

12.9.2 Concurrency Control

Once pre-fetching is complete and the network link is broken, we would encounter the concurrency problems associated with having multiple copies of a file. The obvious approach to concurrency control in this context would be the "checkout" model found in most version control tools and some modern database systems (e.g., [196, 136]). In this model, each pre-fetched file would be locked in either a `read only` or `writable` mode, depending on whether the file is only needed for reading or possibly may be updated during the disconnected process fragment. These locks would be maintained persistently until later reconnection and "checkin".

But a more flexible approach is desirable for some software engineering applications [16]. Fortunately, in addition to being parameterized by the desired process, Oz includes a sophisticated approach to concurrency control whereby new lock modes, compatibility among lock modes, and resolution of locking conflicts can also be defined on a project-specific basis [31, 100]. Locks in Oz are applied to an object in the objectbase, rather than directly on the files associated with an object. These facilities would be exploited to support the LAPUTA disconnected client.

The LAPUTA disconnected client would introduce a new set of persistent locks to Oz's objectbase. Objects which contain read-only files would be locked in a new `dirty read` mode and the files are replicated on the LAPUTA client; unlike `shared` mode, `dirty read` is defined to be compatible with the `exclusive` mode so that other users could continue to work on the file. An obvious example of objects and their associated files that could be locked in `dirty read` mode are "C" header files that a user does not intend to edit, but which are needed to compile a modified "C" source object's file.

[18]Oz contains a flexible statistics gathering package which is capable of computing a wide range of usage statistics about an environment.

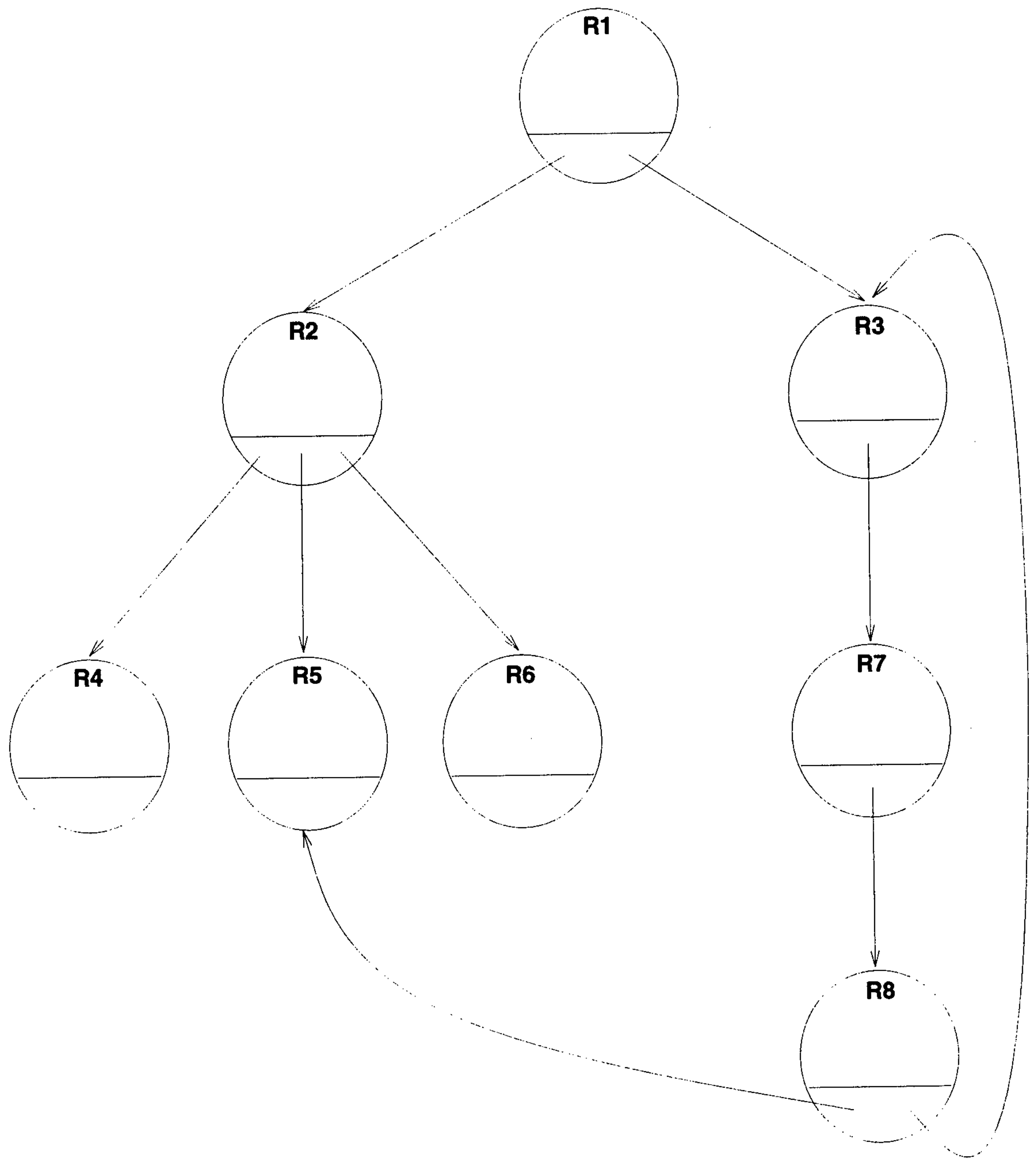


Figure 12.2: Transitive Closure of Rule Set

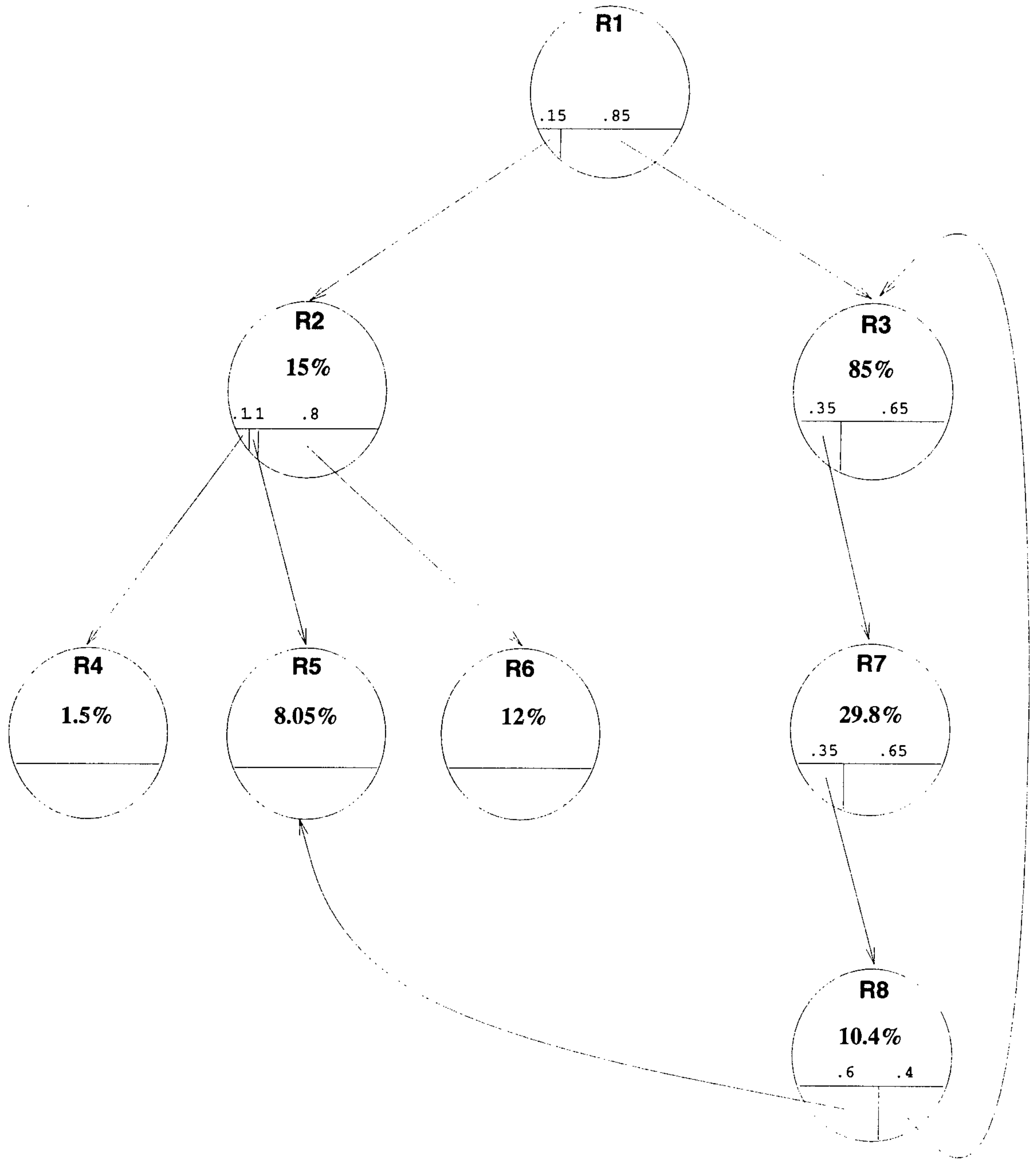


Figure 12.3: Probability Graph

The use of the `dirty read` lock would allow other users to make modifications, and the disconnected user would continue to use the outdated version of the file until reintegration occurred.

Objects which contain write-able files could be locked in one of two modes, `creative exclusive` or `generated exclusive`. The process architect would describe a task as either `creative` or `generated`. A `creative` task is one that involves an interactive tool, usually one that produces a valuable product, such as an editor or a drawing program. These tasks are separated from `generated` tasks whose output can easily be reproduced without direct user involvement. `Generated` tasks typically read in one or more input files, process the input, and produce one or more output files without modifying the inputs. Examples of `generated` tasks are assembling, compiling, and linking because the tools used in these tasks can easily be invoked to re-create their output. If a file were pre-fetched in order to be modified by a `creative` task, the file would be locked in `creative exclusive` mode. Otherwise the file would be locked in `generated exclusive` mode.

The two `exclusive` modes are useful during reintegration. Due to the `dirty read` mode used in LAPUTA, it would be possible for a pre-fetched file to become inconsistent with the version stored in the server's repository. Furthermore, files locked in `generated exclusive` mode may be generated with an inconsistent file. Files from objects locked in `creative exclusive` mode would always be "dominant", i.e., they would always be considered the most recent copy of a file and so could always safely replace older versions in the shared file repository upon reintegration. Because `generated` tasks invoke tools that read input files from objects locked in `dirty read` mode, some caution would need to be exercised when re-integrating these files, to assure that all `generated` files are consistent with their respective input files as found in the repository. The lock matrix shown in figure 12.4 summarizes the compatibility between the various lock modes.

12.9.3 Reintegration

A network connection could be re-established by the disconnected user at any time, at which point reintegration would begin. Reintegration would first detect any changes between the objectbase attributes of objects locked in `dirty read` or `generated exclusive` mode. If no difference were found, the files locked in both `creative exclusive` and `generated exclusive` modes could be presumed valid — and would be copied into the repository, overwriting the previous versions.

But if some shared object attributes had changed, then reintegration would occur in four stages:

1. All files from objects locked in `creative exclusive` mode would be copied into the shared repository, replacing previous versions.

2. All files from objects locked in `generated exclusive` mode that are dependent[19] upon files from objects locked in `dirty read` mode, which in turn are different from the versions stored in the shared repository, would be deleted.

3. All files of objects locked in `generated exclusive` mode, which are dependent upon objects locked in `generated exclusive` mode but which contain files deleted in step 2, would be deleted. This step would iterate through the complete transitive closure.

4. All remaining objects locked in `generated exclusive` mode are presumed to be valid and their files would be copied into the shared repository.

Once all of the files updated in the LAPUTA client had been copied into the main repository, all files which were part of objects locked in `generated exclusive` mode and deleted in steps 2 and 3 could be regenerated by the process. This step would be possible because only files generated by tools (without direct user intervention) would have been deleted. The process engine would therefore be capable of triggering the appropriate tasks to regenerate all of the missing files, finishing reintegration.

[19]A dependency exists between two files, when one file is read in as input to a tool that either produces the second file, or produces a file upon which the second file is dependent.

CX	=	Creative Exclusive
GX	=	Generated Exclusive
DR	=	Dirty Read
S	=	Shared

	CX	GX	DR	S
CX	no	no	yes	no
GX	no	no	yes	no
DR	yes	yes	yes	yes
S	no	no	yes	yes

Figure 12.4: Laputa lock matrix

12.9.4 Dependency Tracking

The reintegration phase of the LAPUTA disconnected client would rely on the ability to detect dependencies between objects. Unfortunately, this information is not readily available from only the process description. To test for object dependency, a virtual copy of the server's objectbase would be created in the server's memory. Objects that contain modified files would be integrated into the virtual objectbase one at a time. After each modified object had been integrated into the virtual objectbase, forward chains would be followed to bring the virtual objectbase into a consistent state. At each stage, the virtual objectbase would be compared against the original objectbase maintained in the server. When an attribute of an object changes following a different object's virtual reintegration, we would note the change, and presume a dependency between the two objects. The virtual reintegration would continues until all object dependencies had been found.

12.10 Related Work

While we know of no other SDE architecture which attempts to minimize network bandwidth, there are numerous applications in other domains which use similar bandwidth reduction techniques to those found in LAPUTA. One such application is World Wide Web browsers modified to run with proxy servers [153]. The goal of a WWW based proxy server is two-fold:

1. Communication through a corporate firewall.

2. Bandwidth reduction via file cache.

The WWW browser is modified to pass all http requests to the WWW proxy rather than directly to the http server. The WWW proxy is responsible for either serving the requested pages out of a local cache (in order to reduce the demand on the wide-area network), or for contacting the http server and downloading the pages on behalf of the WWW browser. In this model, the WWW proxy sits between the WWW browser and the http server as an intermediary. To the WWW browser, the proxy appears to be a http server, and the http server views the proxy as a WWW browser.

There are numerous other examples of applications with proxies, however the proxy model introduced in this research seems to be novel in that the Oz Proxy client is a peer of the Low Bandwidth client which it supports.

Disconnected operation could have been achieved in the Sun Network Software Environment [3]. A user would select a software component to check out, and all of its constituent files were "acquired". The user was then able to work independently on the files in the component. Other users were free to "acquire" the same software component, increasing parallelism. On request or at "reconciliation" time, the system detected any changes in the file repository from the user's workspace, and copied the new versions of the checked out files. A `diff`-like tool assisted the user in merging the updated files with their newer versions. Unfortunately, it is often difficult for a user to correctly identify which files should be placed in each software component, and which components are actually needed. Furthermore, while pre-fetching large software components may enable a user to work in disconnected mode, the extremely weak concurrency control of Sun's NSE often leads to reintegration problems when more than one user has edited a file.

Numerous other SDEs employ some form of checkout model for concurrency control [16], but I know of none that either exploits the software process to assist in the selection of files to be checked out, or that permits disconnected operation.

Considerable research in disconnected file systems achieved through file pre-fetching has been done, but systems such as Ficus [117], Coda [137] and Tait's "File System for Mobile Computing" [220] were unable to draw upon the detailed application semantics inherently available from PCEs such as Oz. Files were selected for pre-fetching either by explicit user selection, or by analyzing prior file access patterns.

12.11 Contributions

The implementation of the LAPUTA low bandwidth model within Oz is complete. A fully functional Proxy client has been written which contains approximately 2,500 lines of C source code. XView and Motif low bandwidth clients have been implemented by modifying the existing Oz XView and Motif clients. The Oz server was adapted to support the modified login protocol employed by the Low Bandwidth client, and to support activity delegation to the Proxy client. The total line count for the LAPUTA low bandwidth implementation within Oz is approximately 9,000 lines of new or modified code, spanning over 30 source files. The implementation is specific to Oz, although the model of operation is applicable to many other SDEs.

The contributions of this work are:

1. A new low bandwidth PCE model.

2. A sample low bandwidth implementation within the Oz PCE.

3. A proposed model for disconnected operation in a PCE.

Bibliography

[1] Hussein M. Abdel-Wahab. XTV. http://www.cs.odu.edu/ waha_cit/XTV.doc/xtv.html.

[2] *Transcending Boundaries: ACM 1994 Conference on Computer Supported Cooperative Work*, Chapel Hill NC, October 1994. ACM Press.

[3] Evan W. Adams, Masahiro Honda, and Terrence C. Miller. Object management in a CASE environment. In *11th International Conference on Software Engineering*, pages 154–163, Pittsburgh PA, May 1989.

[4] D. Agrawal, J. L. Bruno, A. El Abbadi, and V. Krishnaswamy. Relative serializability: An approach for relaxing the atomicity of transactions. In *ACM-SIGMOD/PODS 1994 International Conference on Management of Data*, pages 139–149, Minnesota MN, May 1994. ACM.

[5] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Gunthor, and Mohan U. Kamath. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. In *IFIP WG 8.1 Workgroup Conference on Information Systems Development for Decentralized Organizations*, Trondheim, Norway, August 1995.

[6] A. Argento, C. Bonferini, F. Dematte, and S. Manca. ECMA PCTE, CORBA and ATIS. In *PCTE Newsletter*, pages 20–27, Nanterre Cedex, France, June 1992. Jean-Claude Rault.

[7] John E. Arnold and Steven S. Popovich. Integrating, customizing and extending environments with a message-based architecture. Technical Report CUCS-008-95, Columbia University, Department of Computer Science, September 1994. The research described in this report was conducted at Bull HN Information Systems, Inc. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-008-95.ps.Z.

[8] Michael Baentsch, Georg Molter, and Peter Sturm. WebMake: Integrating distributed software development in a structure-enhanced Web. In *3rd International World-Wide Web Conference*, Darmstadt, Germany, April 1995. Elsevier Science B.V. http://www.igd.fhg.de/www/www95/proceedings/papers/51/WebMake/WebMake.html.

[9] Robert Balzer and K. Narayanaswamy. Mechanisms for generic process support. In David Notkin, editor, *1st ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 21–32, Los Angeles CA, December 1993. Special issue of *Software Engineering Notes*, 18(5), December 1993.

[10] Sergio Bandinelli and Alfonso Fuggetta. Computational reflection in software process modeling: the SLANG approach. In *15th International Conference on Software Engineering*, pages 144–154, Baltimore MD, May 1993. IEEE Computer Society Press.

[11] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Sandro Grigolli. Process enactment in SLANG. In J.C. Derniame, editor, *Software Process Technology Second European Workshop*, number 635 in Lecture Notes in Computer Science. Springer-Verlag, Trondheim, Norway, September 1992.

[12] Roger Barga and Calton Pu. A practical and modular method to implement extended transaction models. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *21st International Conference on Very Large Data Bases*, pages 206–217, Zurich, Switzerland, September 1995.

[13] Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University, February 1992. CUCS-001-92. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-001-92.ps.gz.

[14] Naser S. Barghouti. Supporting cooperation in the MARVEL process-centered SDE. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 21–31, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992. ftp://ftp.psl.cs.columbia.edu/pub/psl/sde92.ps.Z.

[15] Naser S. Barghouti and Gail E. Kaiser. Modeling concurrency in rule-based development environments. *IEEE Expert*, 5(6):15–27, December 1990. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-017-90.tar.

[16] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-425-89.ps.Z.

[17] Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Alexander E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, October 1996.

[18] David R. Barstow, Howard E. Shrobe, and Erik Sandewall (editors). *Interactive Programming Environments*. McGraw-Hill, New York, 1984.

[19] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[20] Noureddine Belkhatir, Jacky Estublier, and Walcelio L. Melo. Adele 2: A support to large software development process. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 159–170, Redondo Beach CA, October 1991. IEEE Computer Society Press.

[21] Israel Ben-Shaul and Gail E. Kaiser. *A Paradigm for Decentralized Process Modeling*. Kluwer Academic Publishers, Boston, 1995. http://www.wkap.nl/kapis/CGI-BIN/WORLD/book.htm?0-7923-9631-6.

[22] Israel Z. Ben-Shaul. An object management system for multi-user programming environments. Master's thesis, Columbia University, Department of Computer Science, April 1991. CUCS-010-91.

[23] Israel Z. Ben-Shaul and George T. Heineman. A 3-level atomicity model for decentralized workflow management system. In Carlo Montangero, editor, *5th European Workshop on Software Process Technology*, volume 1149 of *Lecture Notes in Computer Science*, pages 61–64, Nancy, France, October 1996. Springer-Verlag.

[24] Israel Z. Ben-Shaul, George T. Heineman, Steve S. Popovich, Peter D. Skopp, Andrew Z. Tong, and Giuseppe Valetto. Integrating groupware and process technologies in the OZ environment. In Carlo Ghezzi, editor, *9th International Software Process Workshop: The Role of Humans in the Process*, pages 114–116, Airlie VA, October 1994. IEEE Computer Society Press.

[25] Israel Z. Ben-Shaul and Gail E. Kaiser. Process evolution in the Marvel environment. In Wilhelm Schäfer, editor, *8th International Software Process Workshop: State of the Practice in Process Technology*, pages 104–106, Wadern, Germany, March 1993. Position paper. ftp://ftp.psl.cs.columbia.edu/pub/psl/ispw8.ps.Z.

[26] Israel Z. Ben-Shaul and Gail E. Kaiser. A configuration process for a distributed software development environment. In *2nd International Workshop on Configurable Distributed Systems*, pages 123–134, Pittsburgh PA, March 1994. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-022-93.ps.Z.

[27] Israel Z. Ben-Shaul and Gail E. Kaiser. A paradigm for decentralized process modeling and its realization in the Oz environment. In *16th International Conference on Software Engineering*, pages 179–188, Sorrento, Italy, May 1994. IEEE Computer Society Press. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-024-93.ps.Z.

[28] Israel Z. Ben-Shaul and Gail E. Kaiser. An interoperability model for process-centered software engineering environments and its implementation in Oz. Technical Report CUCS-034-95, Columbia University Department of Computer Science, December 1995. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-034-95.ps.Z.

[29] Israel Z. Ben-Shaul and Gail E. Kaiser. Integrating groupware activities into workflow management systems. In *7th Israeli Conference on Computer Systems and Software Engineering*, pages 140–149, Herzliya, Israel, June 1996. IEEE Computer Society Press. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-002-95.ps.Z.

[30] Israel Z. Ben-Shaul and Gail E. Kaiser. Federating process-centered environments: the Oz experience. *Automated Software Engineering*, 5(1), January 1998. In press. Now available as Columbia University Department of Computer Science, CUCS-006-97, March 1997, ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-006-97.ps.gz.

[31] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. *Computing Systems, The Journal of the USENIX Association*, 6(2):65–103, Spring 1993. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-012-92.ps.Z.

[32] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In Lochovsky and Taylor, editors, *6th Conference on Very Large Databases*, Montreal, Canada, October 1980. Morgan Kaufmann Publishers.

[33] A. Biliris, S. Dar, N. Gehani, H.V. Jagadish, and K. Ramamritham. ASSET: A system for supporting extended transactions. In *1994 ACM SIGMOD International Conference on Management of Data*, pages 44–54, Minneapolis MN, May 1994. ACM Press. Special issue of *SIGMOD Record*, 23(2), June 1994.

[34] Barry Boehm, editor. *10th International Software Process Workshop: Process Support of Software Product Lines*, Ventron, France, June 1996.

[35] Gregory Alan Bolcer and Richard N. Taylor. Endeavors: A process system integration infrastructure. In Wilhelm Schäfer, editor, *4th International Conference on the Software Process: Software Process - Improvement and Practice*, pages 76–89, Brighton, UK, December 1996. IEEE Computer Society. http://www.ics.uci.edu/pub/endeavors.

[36] Christian Bremeau. The PCTE Contribution to Ada Programming Support Environments (APSE). In Fred Long, editor, *Software Engineering Environments International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 151–166, Chinon, France, September 1989. Springer-Verlag.

[37] Omran A. Bukhres, Jiansan Chen, Weimin Du, and Ahmed K. Elmagarmid. InterBase: An execution environment for heterogeneous software systems. *Computer*, 26(8):57–69, August 1993.

[38] Michael J. Carey, David J. Dewitt, Goetz Graefe, David M. Haight, Joek E. Richardson, Daniel T. Schuh, Eugene J. Shekita, and Scott L. Vandenburg. The Exodus extensible DBMS project: An overview. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, chapter 7.3, pages 474–499. Morgan Kaufman, San Mateo CA, 1990.

[39] Melissa Chase and Howard Reubenstein. An assessment of KBSA and a look towards the future. Technical Report RL-TR-92-163, Rome Laboratory, June 1992.

[40] Jiansan Chen, Omran A. Bukhres, and Ahmed K. Elmagarmid. IPL: A multidatabaase transaction specification language. In *13th International Conference on Distributed Computing Systems*, pages 439–448, Pittsburgh PA, May 1993. IEEE Computer Society Press.

[41] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2), 1959.

[42] Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of Extended Transaction Models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.

[43] Geoffrey Clemm and Leon Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, January 1990.

[44] XSoft Marketing Communications. Inconcert: Workflow software from xsoft, October 1995. www.xerox.com/XSoft/DataSheets/InConcert.html.

[45] Reidar Conradi, Espen Osjord, Per H. Westby, and Chunnian Liu. Initial software process management in EPOS. *Software Engineering Journal*, 6(5):275–284, September 1991.

[46] David Cornelius. Xremote: A serial line protocol for x. In *6th Annual X Technical Conference*, January 1992.

[47] Bill Curtis, Marc I. Kellner, and Jim Over. Process modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[48] Michael Cusumano and Richard W. Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press, New York, 1995.

[49] Roger Dannenberg. *Resource Sharing In A Network Of Personal Computers*. PhD thesis, Carnegie Mellon University Department of Computer Science, 1982.

[50] Umesh Dayal, Hector Garcia-Molina, Mei Hsu, Ben Kao, and Ming-Chien Shan. Third generation TP monitors: A database challenge. In *1993 SIGMOD International Conference on Management of Data*, pages 393–398, May 1993. Special issue of *SIGMOD Record*, 22(2), June 1993.

[51] Prasun Dewan, editor. *Special Issue on Collaborative Software*, volume 6:2 of *Computing Systems, The Journal of the USENIX Association.* University of California Press, Spring 1993.

[52] Prasun Dewan and Rajiv Choudhary. A high-level and flexible framework for implementing multiuser user interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, October 1992.

[53] Prasun Dewan and John Riedl. Toward computer-supported concurrent software engineering. *Computer*, 26(1):17–27, January 1993.

[54] Piotr Krychniak Dimitris Georgakopoulos, Mark Hornick and Frank Manola. Specification and management of extended transactions in a programmable transaction environment. In *10th International Conference on Data Engineering*, pages 462–473, Houston TX, February 1994.

[55] Klaus R. Dittrich, Willi Gotthard, and Peter C. Lockemann. Damokles — a database system for software engineering environments. In *Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 353–371. Springer-Verlag, Berlin, 1986.

[56] Stephen E. Dossick and Gail E. Kaiser. WWW access to legacy client/server applications. In *5th International World Wide Web Conference*, pages 931–940, Paris, France, May 1996. Elsevier Science B.V. Special issue of *Computer Networks and ISDN Systems, The International Journal of Computer and Telecommunications Networking*, 28(7-11), May 1996. http://www.psl.cs.columbia.edu/papers/CUCS-003-96.html.

[57] Mark Dowson. ISTAR — an integrated project support environment. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 27–33, Palo Alto, CA, December 1986. Special issue of *SIGPLAN Notices*, 22(1), January 1987.

[58] Mark Dowson. Integrated project support with ISTAR. *IEEE Software*, 4(6):6–15, November 1987.

[59] Michael Elhadad. *Using argumentation to control lexical choice: a unification-based implementation.* PhD thesis, Columbia University, Department of Computer Science, 1993.

[60] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 2nd Edition.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[61] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon A Distributed Transaction Facility.* Morgan Kaufman, San Mateo CA, 1991.

[62] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–632, November 1976.

[63] R. Ahmed *et al.* The Pegasus heterogenous multidatabase system. *Computer*, 24(12):19–27, December 1991.

[64] European Computer Manufacturers Association. *ECMA PCTE C Programming Language Binding*, 1991.

[65] Michael E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, SE-12(7):744–751, July 1986.

[66] Mary F. Fernandez, Stanley B. Zdonik, and Alan N. Ewald. ObServer: A storage system for object-oriented applications. September 1989.

[67] Christer Fernström. PROCESS WEAVER: Adding process support to UNIX. In *2nd International Conference on the Software Process: Continuous Software Process Improvement*, pages 12–26, Berlin, Germany, February 1993. IEEE Computer Society Press.

[68] James C. Ferrans. The PCTE workbench. In *PCTE Newsletter*, pages 12–15, Nanterre Cedex, France, October 1993. Jean-Claude Rault.

[69] James C. Ferrans, David W. Hurst, Michael A. Sennett, Burton M. Covnot, Wenguang Ji, Peter Kajka, and Wei Ouyang. Hyperweb: A framework for hypermedia-based environments. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 149–158, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[70] Rodrigo F. Flores. The value of a methodology for workflow, 1995. http://www.actiontech.com/market/papers/method5.html.

[71] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge MA, 1992.

[72] Jim Fulton and Chris Kent Kantarjiev. An update on low bandwidth x (lbx), a standard for x and serial lines. Technical Report P93-00001, Xerox Palo Alto Research Center, February 1993.

[73] George T. Heineman Gail E. Kaiser, Israel Z. Ben-Shaul and Wilfredo Marrero. Process evolution for constraint-enforcing environments. Technical Report CUCS-047-92, Columbia University Department of Computer Science, March 1994.

[74] F. Gallo, G. Boudier, and I. Thomas. Overview of PCTE and PCTE+. *ACM SIGPLAN Notices*, 24(2), February 1989.

[75] Hector Garcia-Molina and Kenneth Salem. Sagas. In Umeshwar Dayal and Irv Traiger, editors, *ACM SIGMOD Conference on Management of Data*, pages 249–259, San Francisco CA, May 1987. Special issue of *SIGMOD Record*, 16(3), December 1987.

[76] Pankaj K. Garg and Mehdi Jazayeri, editors. *Process-Centered Software Engineering Environments*. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[77] Pankaj K. Garg, Peiwei Mi, Thuan Pham, Walt Scacchi, and Gary Thunquest. The SMART approach for software process engineering. In *16th International Conference on Software Engineering*, pages 341–350, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[78] P.K. Garg, T. Pham, B. Beach, A. Deshpande, A. Ishizaki, K. Wentzel, and W. Fong. Matisse: A knowldge-based team programming environment. *International Journal of Software Engineering and Knowledge Engineering*, 4(1):15–59, 1994.

[79] David Garlan and Ehsan Ilias. Low-cost, adaptable tool integration policies for integrated environments. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 1–10, Irvine CA, December 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[80] Jorge F. Garza and Won Kim. Transaction management in an object-oriented database system. In *SIGMOD International Conference on Data Management*, pages 37–45, Chicago IL, June 1988. Special issue of *SIGMOD Record*, 17(3), September 1988.

[81] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modelling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–152, 1995.

[82] Mari Georges and Claude Koemmer. Use and Extension of PCTE: The SPMMS Information System. In Fred Long, editor, *Software Engineering Environments International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 271–282, Chinon, France, September 1989. Springer-Verlag.

[83] Carlo Ghezzi, editor. *9th International Software Process Workshop: The Role of Humans in the Process*, Airlie VA, October 1994. IEEE Computer Society Press.

[84] GIE Emeraude. *Emeraude PCTE Environment Guide*, 1994.

[85] John Gintell, John Arnold, Michael Houde, Jacek Kruszelnicki, Roland McKenney, and Gerard Memmi. Scrutiny (TM): A collaborative inspection and review systems. In Ian Sommerville and Manfred Paul, editors, *4th European Software Engineering Conference*, number 717 in Lecture Notes in Computer Science, pages 344–360, Garmisch-Partenkirchen, Germany, September 1993. Springer-Verlag.

[86] Mark A. Gisi and Gail E. Kaiser. Extending a tool integration language. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 218–227, Redondo Beach CA, October 1991. IEEE Computer Society Press. ftp://ftp.psl.cs.columbia.edu/pub/psl/icsp91.ps.Z.

[87] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation*. Addison-Wesley, Reading MA, 1983.

[88] J. Gray, R. Lorie, and G. Putzolu. Granularity of locks and degrees of consistency in a shared database. In *International Conference on Very Large Data Bases*, pages 428–451. Morgan Kaufmann, 1975.

[89] Jonathan Grudin. Computer supported cooperative work: History and focus. *Computer*, 27(5), May 1994.

[90] Volker Gruhn and Rudiger Jegelka. An evaluation of FUNSOFT nets. In J.C. Derniame, editor, *Software Process Technology Second European Workshop*, number 635 in Lecture Notes in Computer Science, pages 196–214. Springer-Verlag, Trondheim, Norway, September 1992.

[91] A.N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[92] Thanasis Hadzilacos and Vassos Hadzilacos. Transactions synchronization in object bases. *Journal of Computer and System Sciences*, 43:2–24, 1991.

[93] William Harrison. RPDE^3: A framework for integrating tool fragments. *IEEE Software*, 4(6):46–56, November 1987.

[94] William Harrison, Harold Ossher, and Mansour Kavianpour. OOTIS: extending PCTE with fine-grained tool composition. In *PCTE Newsletter*, pages 11–19, Nanterre Cedex, France, December 1992. Jean-Claude Rault.

[95] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence Journal*, 26:251–321, 1985.

[96] Dennis Heimbigner. Proscription versus Prescription in process-centered environments. In Takuya Katayama, editor, *6th International Software Process Workshop: Support for the Software Process*, pages 99–102, Hakodate, Japan, October 1990. IEEE Computer Society Press.

[97] Dennis Heimbigner. A federated architecture for environments: Take II. In *Process Sensitive SEE Architectures Workshop*, Boulder CO, September 1992. Preprints.

[98] Dennis Heimbigner. The ProcessWall: A process state server approach to process programming. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 159–168, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[99] Dennis Heimbigner and Leon Osterweil. An argument for the elimination of roles. In Carlo Ghezzi, editor, *9th International Software Process Workshop*, Airlie VA, October 1994. IEEE Computer Society Press. In press.

[100] George T. Heineman. A transaction manager component for cooperative transaction models. Technical Report CUCS-017-93, Columbia University Department of Computer Science, July 1993. PhD Thesis Proposal.

[101] George T. Heineman. Automatic translation of process modeling formalisms. In *1994 Centre for Advanced Studies Conference (CASCON)*, pages 110–120, Toronto ON, Canada, November 1994. IBM Canada Ltd. Laboratory. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-036-93.ps.Z.

[102] George T. Heineman. *A Transaction Manager Component for Cooperative Transaction Models.* PhD thesis, Columbia University Department of Computer Science, June 1996. CUCS-010-96. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-010-96.ps.gz.

[103] George T. Heineman and Gail E. Kaiser. Incremental process support for code reengineering. In *International Conference on Software Maintenance*, pages 282–290, Victoria BC, Canada, September 1994. IEEE Computer Society Press.

[104] George T. Heineman and Gail E. Kaiser. Integrating a transaction manager component with Process-WEAVER. Technical Report CUCS-012-94, Columbia University Department of Computer Science, May 1994. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-012-94.ps.Z.

[105] George T. Heineman and Gail E. Kaiser. An architecture for integrating concurrency control into environment frameworks. In *17th International Conference on Software Engineering*, pages 305–313, Seattle WA, April 1995. ACM Press. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-021-94.ps.Z.

[106] George T. Heineman and Gail E. Kaiser. The cord approach to extensible concurrency control. Technical Report WPI-CS-TR-96-1, Worcester Polytechnic Institute, 1996. http://cs.wpi.edu/Resources/techreports.

[107] George T. Heineman and Gail E. Kaiser. The CORD approach to extensible concurrency control. In *Thirteenth International Conference on Data Engineering*, Birmingham, UK, April 1997. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-024-95.ps.gz.

[108] George T. Heineman, Gail E. Kaiser, Naser S. Barghouti, and Israel Z. Ben-Shaul. Rule chaining in MARVEL: Dynamic binding of parameters. *IEEE Expert*, 7(6):26–32, December 1992. ftp://ftp.psl.cs.columbia.edu/pub/psl/expert92.ps.Z.

[109] SynerVision for SoftBench: A Process Engine for Teams, 1992. Marketing literature.

[110] D. Hollinsworth. The workflow reference model. Technical Report TC00-1003, Workflow Management Coalition. http://www.aiai.ed.ac.uk/WfMC/.

[111] Karen E. Huff and Victor R. Lesser. A plan-based intelligent assistant that supports the software development process. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 97–106, Boston MA, November 1988. ACM Press. Special issue of *SIGPLAN Notices*, 24(2), February 1989 and of *Software Engineering Notes*, 13(5), November 1988.

[112] Watts Humphrey and Marc I. Kellner. Software process modeling: Principles of entity process models. In *11th Internation Conference on Software Engineering*, pages 331–342, Pittsburgh PA, May 1989. IEEE Computer Society Press.

[113] *2nd International Conference on the Software Process: Continuous Software Process Improvement*, Berlin, Germany, February 1993. IEEE Computer Society Press.

[114] Hajimu Iida, Takeshi Ogihara, Katsuro Inoue, and Koji Torii. Generating a menu-oriented navigation system from formal description of software development activity sequence. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 45–57, Redondo Beach CA, October 1991. IEEE Computer Society Press.

[115] Van Jacobson and Steve McCanne. wb. van@ee.lbl.gov, mccanne@ee.lbl.gov, The development of 'wb', 'vat', and 'sd' were supported by the Director, Office of Energy Research, Scientific Computing Staff, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

[116] W. Jin, L. Ness, M. Rusinkiewicz, and A. Sheth. Concurrency control and recovery of multi-database work flows in telecommunication applications. In *ACM SIGMOD International Conference on Management of Data*, pages 456–459, Washington DC, May 1993. Special issue of *SIGMOD Record*, 22:2, June 1993.

[117] R.G. Guy J.S. Heideman, T.T. Page and G.J. Popek. Primarily disconnected operation: Experiences with ficus. In *2nd Workshop on Management of Replicated Data*, Monterey CA, November 1992. IEEE Computer Society Press.

[118] Gail E. Kaiser. Cooperative transactions for multi-user environments. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter 20, pages 409–433. ACM Press, New York NY, 1994. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-006-93.ps.Z.

[119] Gail E. Kaiser, Naser S. Barghouti, and Michael H. Sokolsky. Experience with process modeling in the MARVEL software development environment kernel. In Bruce Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, January 1990. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-446-89.ps.gz.

[120] Gail E. Kaiser, Israel Z. Ben-Shaul, and Steven S. Popovich. Implementing activity structures process modeling on top of the MARVEL environment kernel. Technical Report CUCS-027-91, Columbia University, September 1991.

[121] Gail E. Kaiser, Israel Z. Ben-Shaul, Steven S. Popovich, and Stephen E. Dossick. A metalinguistic approach to process enactment extensibility. In Wilhelm Schäfer, editor, *4th International Conference on the Software Process: Improvement and Practice*, pages 90–101, Brighton, UK, December 1996. IEEE Computer Society. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-016-96.ps.gz.

[122] Gail E. Kaiser, Stephen E. Dossick, Wenyu Jiang, and Jack Jingshuang Yang. An architecture for WWW-based hypercode environments. In *1997 International Conference on Software Engineering: Pulling Together*, pages 3–13, Boston MA, May 1997. Association for Computing Machinery. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-037-96.ps.gz.

[123] Gail E. Kaiser and Peter H. Feiler. An architecture for intelligent assistance in software development. In *9th International Conference on Software Engineering*, pages 180–188, Monterey CA, March 1987. IEEE Computer Society Press. ftp://ftp.psl.cs.columbia.edu/pub/psl/icse87.ps.gz.

[124] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-401-88.tar.Z.

[125] Gail E. Kaiser, George T. Heineman, Peter D. Skopp, and Jack J. Yang. Incremental process support for component-based software engineering. Technical Report CUCS-007-96, Columbia University Department of Computer Science, April 1997. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-007-96.ps.gz.

[126] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul. A bi-level language for software process modeling. In *15th International Conference on Software Engineering*, pages 132–143, Baltimore MD, May 1993. IEEE Computer Society Press.

[127] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul. A bi-level language for software process modeling. In Walter F. Tichy, editor, *Configuration Management*, number 2 in Trends in Software, chapter 2, pages 39–72. John Wiley & Sons, 1994. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-016-92.ps.Z.

[128] Gail E. Kaiser and Calton Pu. Dynamic restructuring of transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 8, pages 265–295. Morgan Kaufmann, San Mateo CA, 1992.

[129] Simon Kaplan, editor. *Conference on Organizational Computing Systems*, Milpitas CA, November 1993. ACM Press.

[130] Simon M. Kaplan, William J. Tolone, Alan M. Carroll, Douglas P. Bogia, and Celsina Bignoli. Supporting collaborative software development with ConversationBuilder. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 11–20, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[131] Takuya Katayama. A hierarchical and functional software process description and its enaction. In *11th International Conference on Software Engineering*, pages 343–352, Pittsburgh PA, May 1989. IEEE Computer Society Press.

[132] Marc I. Kellner and H. Dieter Rombach. Session summary: Comparisons of software process descriptions. In Takuya Katayama, editor, *6th International Software Process Workshop: Support for the Software Process*, pages 7–18, Hakodate, Japan, October 1990. IEEE Computer Society Press.

[133] Brian W. Kernighan and John R. Mashey. The UNIX programming environment. *Computer*, 12(4):25–34, April 1981. Reprinted in [18].

[134] Kenyon Brown Kevin Brown and Kyle Brown. *Mastering LOTUS NOTES.* SYBEX, San Francisco CA, 1995.

[135] N. Kiesel, A. Schurr, and B. Westfechtel. GRAS, a graph-oriented database system for software engineering applications. In Hing-Yang Lee, Thomas F. Reid, and Stan Jarzabek, editors, *6th International Workshop on Computer-Aided Software Engineering*, pages 272–286, Singapore, July 1993.

[136] Won Kim, Nat Ballou, Jorge F. Garz, and Darrell Woelk. A Distributed Object-Oriented Database System Supporting Shared and Private Databases. *ACM Transactions on Information Systems*, 9(1):31–51, January 1991.

[137] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *13th ACM Symposium on Operating System Principles*, pages 213–225. ACM, October 1991.

[138] Henry F. Korth. Extending the scope of relational languages. *IEEE Software*, 3(1):19–28, January 1986.

[139] Henry F. Korth. The double life of the transaction abstraction: Fundamental principles and evolving system concepts. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *21st International Conference on Very Large Data Bases*, pages 2–6, Zurich, Switzerland, September 1995. Invited paper.

[140] Balachander Krishnamurthy and Naser S. Barghouti. Provence: A process visualization and enactment environment. In Ian Sommerville and Manfred Paul, editors, *4th European Conference on Software Engineering*, volume 717 of *Lecture Notes in Computer Science*, pages 151–160. Springer-Verlag, Garmisch-Partenkirchen, Germany, September 1993.

[141] H. T. Kung and John Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[142] Programming Systems Lab. Marvel 3.1.1 administrator manual. Technical Report CUCS-038-93c, Columbia University Department of Computer Science, 1993.

[143] Programming Systems Lab. *Amber Manual*, July 1996. ftp://ftp.psl.cs.columbia.edu/pub/psl/oz.1.2.manuals/V.Amber/.

[144] Programming Systems Lab. *Darkover Manual*, July 1996. ftp://ftp.psl.cs.columbia.edu/pub/psl/oz.1.2.manuals/IV.Darkover_API/.

[145] David B. Leblang and Robert P. Chase, Jr. Parallel software configuration management in a network environment. *IEEE Software*, 4(6):28–35, November 1987.

[146] Jintae Lee, Gregg Yost, and the PIF Working Group. The PIF process interchange format and framework, December 1994. http://www-sloan.mit.edu/ccs/pifmain.html.

[147] Wenke Lee and Gail E. Kaiser. Interfacing Oz with the PCTE OMS. Technical Report CUCS-012-95, Columbia University, Department of Computer Science, June 1997.

[148] Wenke Lee, Gail E. Kaiser, Paul D. Clayton, and Eric H. Sherman. OzCare: A workflow automation system for care plans. *Journal of the American Medical Informatics Association: 1996 AMIA Annual Fall Symposium*, pages 577–581, October 1996. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-012-96.ps.Z.

[149] Ted G. Lewis. Where is Client/Server software headed? *Computer*, 28(4):49–55, April 1995.

[150] F. Leymann and D. Roller. Business processes management with FlowMark. In *39th IEEE Computer Society International Conference (CompCon), Digest of Papers*, pages 230–233, San Francisco, March 1994.

[151] Chunnian Liu. Software process planning and execution: Coupling vs. integration. In *Advanced Information Systems Engineering: 3rd International Conference CAiSE '91*, number 498 in Lecture Notes in Computer Science, pages 356–374. Springer-Verlag, Trondheim, Norway, May 1991.

[152] Fred Long and Ed Morris. An overview of PCTE: A basis for a portable common tool environment. Technical Report CMU/SEI-93-TR-1, ESC-TR-93-175, Software Engineering Institute, Carnegie Mellon University, March 1993.

[153] Ari Luotonen and Kevin Altis. World-Wide Web Proxies
. http://www.w3.org/hypertext/WWW/Proxies/, April 1994.

[154] Nazim H. Madhavji and Maria H. Penedo, editors. *Special Section on the Evolution of Software Processes*, volume 19:12 of *IEEE Transactions on Software Engineering*. December, 1993.

[155] Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring parallel and distributed progrmas. *Software Engineering Journal*, 8(2):73–82, March 1993.

[156] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[157] Workflow Management Coalition Members. Coalition overview, September 1995. http://www.aiai.ed.ac.uk/WfMC/overview.html.

[158] Workflow Management Coalition Members. Workflow management coalition reference model, February 1995. www.aiai.ed.ac.uk/WfMC/GIF/WfMC-ref-model.GIF.

[159] Peiwei Mi and Walt Scacchi. A knowledge-based environment for modeling and simulating software engineering processes. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):283–294, September 1990.

[160] Peiwei Mi and Walt Scacchi. Modeling articulation work in software engineering processes. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 188–201, Redondo Beach CA, October 1991. IEEE Computer Society Press.

[161] Peiwei Mi and Walt Scacchi. Process integration in case environments. *IEEE Software*, 9(2):45–53, March 1992.

[162] Naftaly H. Minsky. Law-governed systems. *Software Engineering Journal*, 6(5):285–302, September 1991.

[163] C. Mohan, D. Haderle, B. Lindsay, H. Piradesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 1991. To Appear. Also IBM Report RJ 6649, Revised 11/90.

[164] Robert Munck, Patricia Oberndorf, Erhard Ploedereder, and Richard Thall. An overview of the DOD-STD-1838A (proposed), the Common APSE Interface Set, revision A. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 235–247, Boston MA, November 1988. ACM Press. Special issues of *Software Engineering Notes*, 13(5), November 1988 and *SIGPLAN Notices*, 24(2), February 1989.

[165] Erich Neuhold and Michael Stonebraker (editors). Future directions in DBMS research. *SIGMOD Record*, 18(1):17–26, March 1989.

[166] John R. Nicol, C. Thomas Wilkes, and Frank A. Manola. Object orientation in heterogeneous distributed computing systems. *Computer*, 26(6):57–67, June 1993.

[167] John Niles. Beyond telecommuting: New paradigm for effect of telecommunications on travel. http://www.lbl.gov:80/ICSD/Niles/.

[168] Reference Model for Frameworks of Software Engineering Environments: Edition 3 of Technical Report ECMA TR/55, August 1993. NIST Special Publication 500-211. Available as `/pub/isee/sp.500-211.ps` via anonymous ftp from nemo.ncsl.nist.gov.

[169] David Notkin and William G. Griswold. Extension and software development. In *10th International Conference on Software Engineering*, pages 274–283, Raffles City, Singapore, April 1988. IEEE.

[170] Jacky Estublier Noureddine Belkhatir and Walcelio L. Melo. Software process model and work space control in the adele system. In *2nd International Conference on the Software Process: Continuous Software Process Improvement*, pages 2–11, Berlin Germany, February 1993. IEEE Computer Society Press.

[171] Object Management Group. http://www.omg.org/.

[172] Corba 2.0/iiop specification. Technical Report formal/97-02-25, Object Management Group, Framingham MA, 1997. http://www.omg.org/corba/corbiiop.htm.

[173] Harold Ossher and William Harrison. Support for change in RPDE^3. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 218–228, Irvine CA, December 1990. ACM Press. Special issue of *SIGSOFT Software Engineering Notes*, 15(6), December 1990.

[174] Leon J. Osterweil. Presentation at Software Process Architectures Workshop, March 1995.

[175] John K. Ousterhout. Tcl: An embeddable command language. In *Winter 1990 USENIX Conference*, pages 133–146, Washington DC, January 1990. USENIX Association.

[176] Maria H. Penedo. Life-cycle (sub) process scenario. In Carlo Ghezzi, editor, *9th International Software Process Workshop*, pages 141–143, Airlie VA, October 1994. IEEE Computer Society Press.

[177] D. Perkins. RFC 1171: The Point-to-Point Protocol for the Transmission of Multi-Protocol Datagrams Over Point-to-Point Links, July 1990.

[178] James L. Peterson. *Petri Net Theory and The Modeling of Systems.* Prentice-Hall, Englewood Cliffs NJ, 1981.

[179] Burkhard Peuschel and Wilhelm Schäfer. Concepts and implementation of a rule-based process engine. In *14th International Conference on Software Engineering*, pages 262–279, Melbourne Australia, May 1992. Association for Computing Machinery.

[180] Burkhard Peuschel and Stefan Wolf. Architectural support for distributed process centered software development environments. In Wilhelm Schäfer, editor, *8th International Software Process Workshop: State of the Practice in Process Technology*, pages 126–128, Wadern, Germany, March 1993. IEEE Computer Society Press. Position paper.

[181] Steven S. Popovich. Rule-based process servers for software development environments. In John Botsford, Arthur Ryman, Jacob Slonim, and David Taylor, editors, *1992 Centre for Advanced Studies Conference (CASCON)*, volume I, pages 477–497, Toronto ON, Canada, November 1992. IBM Canada Ltd. Laboratory.

[182] Steven S. Popovich. *An Architecture for Extensible Workflow Process Servers.* PhD thesis, Columbia University Department of Computer Science, January 1997. CUCS-014-96. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-014-96.ps.gz.

[183] Steven S. Popovich and Gail E. Kaiser. Integrating an existing environment with a rule-based process server. Technical Report CUCS-004-95, Columbia University Department of Computer Science, August 1995. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-004-95.ps.Z.

[184] Programming Systems Laboratory. *Oz 1.2 Manual Set*, July 1996. Columbia University Department of Computer Science. ftp://ftp.psl.cs.columbia.edu/pub/psl/oz.1.2.manuals.

[185] CLF Project. *CLF Manual,*. USC Information Sciences Institute, January 1988.

[186] Calton Pu. Superdatabases for composition of heterogeneous databases. In Amar Gupta, editor, *Integration of Information Systems: Bridging Heterogeneous Databases*, pages 150–157. IEEE Press, 1989. Also appeared in *4th International Conference on Data Engineering*, Los Angeles CA, 1988.

[187] Calton Pu. Generalized transaction processing with Epsilon-Serializability. In *1991 International Workshop on High Performance Transaction Systems*, 1991.

[188] James M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[189] James M. Purtilo and Christine R. Hofmeister. Dynamic reconfiguration of distributed programs. In *11th International Conference on Distributed Computing Systems*, pages 560–571, Arlington TX, May 1991. IEEE Computer Society Press.

[190] Sudha Ram, editor. *Special Issue on Heterogeneous Distributed Database Systems*, volume 24:12 of *Computer*. IEEE Computer Society Press, December 1991.

[191] Marcus J. Ranum. A network firewall. In *World Conference on Systems Management and Security*, 1992.

[192] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, Palo Alto CA, October 1987. Special issue of *SIGPLAN Notices*, 22(10), October 1987.

[193] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[194] Steven P. Reiss. *THE FIELD PROGRAMMING ENVIRONMENT: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, Boston, 1995.

[195] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1989.

[196] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1:364–370, 1975.

[197] J. Romkey. RFC 1055: Nonstandard for transmission of IP datagrams over serial lines: SLIP, June 1988.

[198] David S. Rosenblum and Balachander Krishnamurthy. An event-based model of software configuration management. In Peter H. Feiler, editor, *3rd International Workshop on Software Configuration Management*, pages 94–97. ACM Press, June 1991.

[199] Kenneth Salem, Hector Garcia-Molina, and Rafael Alonso. Altruistic locking: A strategy for coping with long lived transactions. In *2nd International Workshop on High Performance Transaction Systems*, Pacific Grove CA, September 1987.

[200] Wilhelm Schäfer, editor. *4th International Conference on the Software Process*, Brighton, UK, December 1996. IEEE Computer Society Press.

[201] Wilhelm Schäfer, Burkhard Peuschel, and Stefan Wolf. A knowledge-based software development environment supporting cooperative work. *International Journal on Software Engineering & Knowledge Engineering*, 2(1):79–106, March 1992.

[202] Friedemann Schwenkreis. Workflow for the German Federal Government. In *NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, pages 64–68, Athens GA, May 1996. Position paper.

[203] Richard W. Selby, Adam A. Porter, Doug C. Schmidt, and Jim Berney. Metric-driven analysis and feedback systems for enabling empirically guided software development. In *13th International Conference on Software Engineering*, pages 288–298, Austin TX, May 1991. IEEE Computer Society Press.

[204] Amit Sheth, Dimitrios Georgakopoulos, Stef M.M. Joosten, Marek Rusinkiewicz, Walt Scacchi, Jack Wileden, and Alexander Wolf. Report from the NSF workshop on workflow and process automation in information systems. Technical Report UGA-CS-TR-96-003, University of Georgia Department of Computer Science, October 1996. http://lsdis.cs.uga.edu/activities/NSF-workflow/final-report-cover.html/test.html.

[205] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[206] Yoichi Shinoda and Takuya Katayama. Towards formal description and automatic generation of programming environments. In Fred Long, editor, *Software Engineering Environments International Workshop on Environments*, number 467 in Lecture Notes in Computer Science, pages 261–270. Springer-Verlag, Chinon, France, September 1989.

[207] Izhar Shy, Richard Taylor, and Leon Osterweil. A metaphor and a conceptual framework for software development environments. In Fred Long, editor, *Software Engineering Environments International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 77–97, Chinon, France, September 1989. Springer-Verlag.

[208] Andrea Helen Skarra. *A Model of Concurrency Control for Cooperating Transactions.* PhD thesis, Brown University, May 1991.

[209] Peter D. Skopp. Low bandwidth operation in a multi-user software development environment. Master's thesis, Columbia University Department of Computer Science, December 1995. CUCS-035-95. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-035-95.ps.Z.

[210] Peter D. Skopp and Gail E. Kaiser. Disconnected operation in a multi-user software development environment. In Bharat Bhargava, editor, *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 146–151, Princeton NJ, October 1993. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-026-93.ps.Z.

[211] Richard Snodgrass and Karen Shannon. Supporting flexible and efficient tool integration. In Tor M. Didriksen Reidar Conradi and Dag H. Wanvik, editors, *Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 290–313. Springer-Verlag, Trondheim, Norway, 1986.

[212] Richard Snodgrass and Karen Shannon. Fine grained data management to achieve evolution resilience in a software development environment. In Richard N. Taylor, editor, *SIGPLAN '90 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 144–156, Irvine CA, December 1990. Special issue of *SIGSOFT Software Engineering Notes*, 15(6), December 1990.

[213] Michael H. Sokolsky and Gail E. Kaiser. A framework for immigrating existing software into new software development environments. *Software Engineering Journal*, 6(6):435–453, November 1991.

[214] Richard Mark Soley and William Kent. The OMG object model. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter 2, pages 18–41. ACM Press, New York NY, 1994.

[215] E. Solomita, J. Kempf, and D. Duchamp. Xmove: A pseudoserver for X window movement. *The X Resource*, 1(11):143–170, July 1994.

[216] Richard M. Stallman. Emacs the extensible, customizable, self-documenting display editor. In *SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–156. ACM, June 1981. Special issue of *SIGPLAN Notices*, 16(6), June 1981.

[217] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. *Hermes A Language for Distributed Computing.* Prentice-Hall, Englewood Cliffs NJ, 1991.

[218] Kevin J. Sullivan and David Notkin. Reconciling environment integration and component independence. In Richard N. Taylor, editor, *SIGSOFT '90 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 22–33, Irvine CA, December 1990. ACM Press. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[219] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. APPL/A: A language for software process programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286, July 1995.

[220] Carl D. Tait and Dan Duchamp. Detection and exploitation of file working sets. In *11th International Conference on Distributed Computing Systems*, pages 2–9, Arlington TX, May 1991. IEEE Computer Society Press.

[221] Ian Thomas. PCTE interfaces: Supporting tools in software-engineering environments. *IEEE Software*, 6(6):15–23, November 1989.

[222] Ian Thomas. Tool Integration in the PACT Environment. In *11th International Conference on Software Engineering*, pages 13–22, Pittsburgh PA, May 1989. IEEE Computer Society Press.

[223] Ian Thomas and Brian A. Nejmeh. Definitions of tool integration for environments. *IEEE Software*, 9(2):29–35, March 1992.

[224] Andrew Z. Tong, Gail E. Kaiser, and Steven S. Popovich. A flexible rule-chaining engine for process-based software engineering. In *9th Knowledge-Based Software Engineering Conference*, pages 79–88, Monterey CA, September 1994. IEEE Computer Society Press. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-005-94.ps.Z.

[225] Encina product overview, 1993. http://www.transarc.com/afs/transarc.com/public/www/Public/Product/Encina/ENCoverview.html.

[226] Giuseppe Valetto and Gail E. Kaiser. Enveloping sophisticated tools into computer-aided software engineering environments. In *IEEE 7th International Workshop on Computer-Aided Software Engineering*, pages 40–48, Toronto Ontario, Canada, July 1995.

[227] Giuseppe Valetto and Gail E. Kaiser. Enveloping sophisticated tools into process-centered environments. *Journal of Automated Software Engineering*, 3:309–345, 1996. ftp://ftp.psl.cs.columbia.edu/pub/psl/CUCS-022-95.ps.gz.

[228] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[229] Kurt Wallnau, Fred Long, and Anthony Earl. Toward a distributed, mediated architecture for workflow management. In *NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, pages 74–84, Athens GA, May 1996. Position paper.

[230] A. I. Wasserman. Tool Integration in Software Engineering Environments. In Fred Long, editor, *Software Engineering Environments: International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 137–149, Chinon, France, September 1989. Springer-Verlag.

[231] Aaron R. Watters. The what, why, who, and where of Python. *UnixWorld Online*, October 1995. Tutorial Article No. 005.

[232] Jeanine Weissenfels, Dirk Wodtke, Gerhard Weikum, and Angelika Kotz-Dittrich. The Mentor architecture for enterprise-wide workflow management. In *NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, pages 69–73, Athens GA, May 1996. Position paper.

[233] David L. Wells, Jose A. Blakeley, and Craig W. Thompson. Architecture of an open object-oriented database management system. *Computer*, 25(10):74–82, October 1992.

[234] Gio Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, March 1992.

[235] Patrick S. Young and Richard N. Taylor. Human-executed operations in the teamware process programming system. In Carlo Ghezzi, editor, *9th International Software Process Workshop: The Role of Humans in the Process*, pages 78–81, Airlie VA, October 1994. IEEE Computer Society Press. Position paper.

[236] Patrick Scott Chun Young. *Customizable Process Specification and Enactment for Technical and Non-Technical Users*. PhD thesis, University of California Irvine, March 1994.

☆U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61201

DISTRIBUTION LIST

addresses	number of copies
JAMES MILLIGAN AFRL/IFTD 525 BROOKS ROAD ROME NEW YORK 13441-4505	5
COLUMBIA UNIVERSITY DEPARTMENT OF COMPUTER SCIENCE NEW YORK, NEW YORK 10027	2
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: MR. JOHN SALASIN DARPA 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1