2

RADC-TR-90-347
Final Technical Report
December 1990

TRANSACTION GRAPHS: A SKETCH FORMALISM FOR ACTIVITY COORDINATION

Software Options, Inc.

Michael Karr

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-347 has been reviewed and is approved for publication.

APPROVED: Douglas A. White

DOUGLAS A. WHITE
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Igor G. Plonisch

IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway Suite 1204 Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE December 1990	3. REPORT TYPE AND DATES COVERED Final Sep 88 to Mar 90

4. TITLE AND SUBTITLE
TRANSACTION GRAPHS: A SKETCH FORMALISM FOR ACTIVITY COORDINATION

5. FUNDING NUMBERS
C F~~D~~30602-87-D-0092
PE - 63728F
PR - 2532
TA - QB
WU - 04

6. AUTHOR(S)
Michael Karr

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Options, Inc.
22 Hilliard St
Cambridge MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-347

11. SUPPLEMENTARY NOTES
RADC Project Engineer: Douglas A. White/COES/(315) 330-3564

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A primary objective of the Knowledge-Based Software Assistant is the automated coordination of all activities comprising the software development process. This automated activities coordination will provide support for managing communications and enforcing policy in software development projects while at the same time enabling automation of the software process.

The purpose of transaction graphs is to serve as a formal basis for an implementation of an activity coordination system. The essential idea is that a transaction graph specifies a distributed computation, which serves as a microcosm of the real-world activities being coordinated. The nodes of the graph correspond to activities, and arcs serve to specify interactions between activities.

Transaction graphs are closed under certain operations and are composable in natural ways. These properties make them a suitable foundation for the detailed design and implementation of tools that aid in coordination. This report addresses how the form alism can be applied to issues such as user interfaces to the system, intuitive means of specifying patterns of coordination.

14. SUBJECT TERMS
Formalism, activity coordination, communication protocol, process program

15. NUMBER OF PAGES
52

16. PRICE CODE

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

Contents

1 Introduction

The goal of this report is to present a formalism which will provide the basis for an implementation of a system for activity coordination. Our intent here is to provide the intellectual groundwork and architecture for the system, rather than a detailed design. In this introduction we will discuss the system criteria which guided the development of the formalism, and in subsequent sections, we present the formalism itself and show how a system based on it can satisfy criteria for the system.

The overriding requirement for an activity coordination system is that it be integrated into the normal everyday activities of the people whose activities are being coordinated. The first consequence of this requirement is that the system must, at a fundamental level, be *distributed.* Because our intended users are national organizations, "distributed" in this setting means not only on local area networks (where communication is essentially continuously available), but also on networks where communication is more sporadic (e.g., by occasional dialup). In particular, *the system cannot rely on any centralized execution component.*

A second consequence of the requirement that the system be integrated into the normal activities of its users is that the system must be *extensible.* We assume that the normal activities of its users involve contact with computers, but we cannot assume much more than that—different user communities will use different machines and different software. If a single activity coordination system is to be of widespread use, it must be possible to connect it to a variety of existing software: without a connection, the use of that software remains unaffected and thus uncoordinated. On the other hand, it is clearly not desirable for the activity coordination system to duplicate functionality of existing software. Rather, the system might control access to such software, might supply some of its input, and might look at its output to assist users in subsequent parts of an activity. It is in making such connections that extensibility is a prerequisite—a closed system simply will not be able to supply the necessary degree of integration.

If an activity coordination system is to be successfully integrated into day-to-day activities, it must be *intuitive*, i.e., easily understood by non-technical users. While there are many factors involved in making a system intuitive, we feel that a graphical aspect to the system is one of the most important. We use the word "graphical" here in two ways: in the mathematical sense, as a structure with nodes and arcs, and in the computer system sense, as a bit-mapped user interface. The use of graphs to describe coordination is quite common: pert charts, organizational charts, and communication networks are examples familiar to people who are not computer experts. And the use of graphics by the best computer interfaces scarcely needs mention.

It would be highly desirable if the extensibility of the system could be accomplished, at least in part, at a very intuitive, presumably graphical, level. This will not be possible at the low level of connecting existing software to the system, but it should be possible for non-experts to connect existing pieces together to provide graph-based high-level descriptions of activity for participants of the system, who could in turn understand "what is happening" in terms of these graphs, and moreover, to be able to interact with the system via such graphs, when this makes sense. Thus, we are proposing that an activity coordination system must have a linguistic component that provides for distributed programs and yet is easy to understand and use! We are aware that this is a difficult, if not impossible, goal, and it is only by restricting the linguistic component to the very highest level that we have a chance of fulfilling it.

Another point to be emphasized about the linguistic component is the necessity of the composability of the constructs and of their parameterizability. Without these classic concepts from programming languages, it is impossible to have the modularity that makes it possible to reuse existing components and to build large systems.

In addition to the general characteristics of distributedness, extensibility, and intuitiveness, there are several technical issues specific to the arena of activity coordination that pervade the design of a system and thus the underlying formalism:

- user-interface—How does a user find out what is going on, and what is to be done next? How is this related to the description of the coordination?
- history—How did the project get into its present state? How is the viewing of history related to the user interface (which usually views the present)?
- simulation—How might things go from here?
- cutover—Descriptions of activities change while the activities themselves are under way. How can this be managed?
- summarization—No single person wants (or may be allowed) to see all of the information known to the activity coordination system. How can information be summarized for presentation to the user or archiving as history?

In this report, we shall discuss transaction graphs, a formalism that provides a framework in which useful solutions to these problems may be tailored for specific applications.

2 Basics

Before giving the semantics of transaction graphs, it is necessary to define their structure. But in understanding the structure, it is helpful to have a glimpse of the key semantic idea:

- using an undirected graph with largely independent computations at the nodes (corresponding to "local" activities),
- coupled with transactions along the arcs (corresponding to the ways in which the activities interact).

The distributed computation of an executing transaction graph is a machine-based microcosm of the real-world activities being coordinated. It guides the users, records their actions, and presents information about the state of the world.

There are several important features which distinguish transaction graphs from schemes like CSP (communicating sequential processes) [Hoa84]. First, the state of each of the computations is visible in a controlled way—the visibility of state provides the basis for presenting information to the users of the system. Second, the use of transactions (which will be explained in section 2.2) seems to provide a better basis for activity coordination than mere message passing, because it more succinctly constrains the behavior of the involved computations. Further, limiting the transactions to those that take place in a specified graph guarantees a simple, intuitive means of depicting patterns of communication.

2.1 Structure

We begin the discussion of the structure of a transaction graph with the definition of a *signature*, which is written a bit like a procedure header:

- $(n_1 : p_1, \ldots, n_k : p_k)$
  - The n_i are distinct symbols.
  - The p_i are *protocols*, drawn from a set P.

As we will see, such a signature describes a node, and each $n_i : p_i$ is associated with one of its k adjacent arcs. The symbol n_i is the name of the arc relative to the node, and p_i describes behavior on the arc from that node's point of view. We will assume a "transposition" operation which takes a protocol and produces a protocol for the other end of the arc:

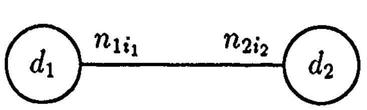
- ${}^T : P \rightarrow P$, where $p^{TT} = p$, for all $p \in P$.

Perhaps the simplest example of a protocol set is one that directs an arc: $P = \{0, 1\}$, where 0 means an incoming directed arc and 1 means an outgoing directed arc. In this case, $0^T = 1$. Another example of a set of protocols can be constructed from the set of types U, where we let $P \stackrel{\text{def}}{=} U \times U$, i.e., a protocol is a pair of types, $\langle t_i, t_i' \rangle$, meaning that the node will make visible a value of type t_i, and will see a value of type t_i' made visible by its neighbor along that arc. In this example, we naturally have $\langle t_i, t_i' \rangle^T = \langle t_i', t_i \rangle$. As we will see, protocols may also specify aspects of the behavior along the arc (as the name "protocol" suggests), not just the types of values.

A *transaction graph* is an ordinary undirected graph, except that it is allowed to have dangling arcs (ones that do not attach to another node). Such a graph is labeled in the following way:

- Each node is labeled with an *activity description*, consisting of several components, one of which is a signature.

- Each non-dangling arc is labeled with two names, one each from the signatures associated with the nodes at its ends. Pictorially, the name appears near the node whose activity description has the signature in which it appears.



- Each dangling arc is also labeled with two names, one from the signature of the activity description on the node to which the arc is attached (this name appears near the node), and another "graph-relative" name, appearing at the dangling end of the arc.

- At every node, each name in the signature of the activity description appears exactly once as the "near" label of an incident arc.

- A graph-relative name appears on only one (dangling) arc.

- Let n_1 and n_2 be the labels on a non-dangling arc, and let $n_i : p_i$ appear in the respective signatures. Then $p_1 = p_2^T$ (equivalently, $p_1^T = p_2$). This is called the *protocol conformance rule.*

The last rule here is analogous to a rule found in many ordinary programming languages for type conformance between actual and formal parameters of a function. Continuing our example of protocols as pairs of types, if $p_i = \langle t_i, t_i' \rangle$, then the protocol conformance rule says that $t_1 = t_2'$ and $t_1' = t_2$.

We close this section by introducing some notation and a definition that will be used throughout.

- Nodes will usually be denoted by the letter u, and arcs by the letter a; in each instance, subscripts are common.

- If arc a is incident on the node u, the name on a near u is $n_{u,a}$, the corresponding protocol from the signature of the activity description on u is $p_{u,a}$, and the opposite end of a from u will be denoted u/a.

- If a is a dangling arc, then u_a is the (unique) node upon which a is incident, and n_a is the graph-relative name on a.

- The *signature of a graph* is defined to be $(n_a : p_{u_a,a})_a$, where a ranges over all dangling arcs.

2.2 Execution

"Execution" will be defined for transaction graphs with no dangling arcs. The first step is to describe another component of an activity description d (in addition to its signature):

- An activity description d has an associated set of states $\mathcal{S}_d$.
- In a transaction graph, we shall use the convention that $\mathcal{S}_u$ denotes $\mathcal{S}_d$ where d is the label of u.
- The set of states for a transaction graph is defined to be $\times_u \mathcal{S}_u$, where u ranges over the nodes of the graph.

An *execution* of a transaction graph consists of a sequence of states for the graph, constrained by rules given below. The rules involve not only states, but elements of the following:

- There is a set of values V that are seen along arcs.

Each of the next two components of an activity description is a set of functions involving $\mathcal{S}_d$ and V. The first:

- Let activity description d have signature $(n_i : p_i)_i$. There are functions:

$$\sigma_{d,i} : \mathcal{S}_d \to V$$

Intuitively, $\sigma_{d,i}$ reveals part of the state of a node labeled with d to the i^{th} neighbor.

The second set of functions formalizes what happens at a transaction. In essence, a node can change the value it makes visible along *one* arc at a time, based on its own state and that of its neighbor along the arc. In conjunction with this operation, it can change its own state. For reasons we discuss below, the state change is non-deterministic.

- Let d have signature $(n_i : p_i)_i$. There are functions:

$$\tau_{d,i} : \mathcal{S}_d \times V \to 2^{\mathcal{S}_d} \text{ where for all } s, v, s' \in \tau_{d,i}(s,v) \text{ and } j \neq i : \sigma_{d,j}(s) = \sigma_{d,j}(s')$$

Intuitively, $\tau_{a,i}$ looks at the node's own state (in $\mathcal{S}_d$) and the value (in V) exposed by the neighbor along the i^{th} arc, and comes up with a new state (an element of $2^{\mathcal{S}_d}$). The condition says that the state change can affect only the value along the arc where the transaction occurs.

The final component of an activity description corresponds to an *out-of-graph* state change, i.e., one not involving a transaction. Out-of-graph state changes correspond either to real world events like the passage of time, or to changes not modeled at a particular level of graph structure.

- Let d have signature $(n_i : p_i)_i$. There is a function:

$$o_d : \mathcal{S}_d \to 2^{\mathcal{S}_d} \text{ where for all } s, s' \in o_d(s) \text{ and } i\text{: } \sigma_{d,i}(s) = \sigma_{d,i}(s')$$

The condition says that exposed values do not change without a transaction, i.e., an application of $\tau_{d,i}$ for some i.

Before we can proceed to the definition of execution, there is one last bit of groundwork, concerning how protocols play a role. The idea is that a protocol is a validation predicate on the sequence of values seen along an arc. Since values seen along the arc can appear at either end, we formalize the sequence as pairs $\langle\langle f_i, v_i\rangle\rangle_i$, where $f_i \in \{0,1\}$; f_i intuitively says which end the value v_i is exposed on. By convention, $f_i = 0$ means that v_i is exposed by σ_i, and thus $f_i = 1$ means that v_i is seen by τ_i. Using S^* to denote the set of all sequences of elements of a set S, we require that:

- An element of P is a predicate on $(\{0,1\} \times V)^*$.

The spirit of transaction graphs is that values are *seen* along arcs, not necessarily transmitted, and in this spirit, an element of P is allowed to pass judgment only on the sequence of *changes* to exposed values. This can be technically stated as follows:

- Let $\langle\langle f_i, v_i\rangle\rangle_i \in (\{0,1\} \times V)^*$, and suppose there are some i_1 and i_2 with $f_i \neq f_{i_1}$ for $i_1 < i \leq i_2$, i.e., there is no change in the value seen at the f_{i_1} end of the arc. Then repeating f_{i_1}, v_{i_1} after changes at the f_{i_2} end of the arc does not affect the value of the protocol, or technically, every $p \in P$ must satisfy:

  $$p(\langle\langle f_i, v_i\rangle\rangle_i) \Leftrightarrow p(\langle\langle f_i, v_i\rangle\rangle_i \frown \langle\langle f_{i_1}, v_{i_1}\rangle\rangle)$$

  In practice, we define the action of a protocol as a predicate only for *reduced* sequences, by which we mean sequences with the property that if i_1 and i_2 are successive indices with the same f_i, then $v_{i_1} \neq v_{i_2}$.

The action of $p \in P$ as a predicate must have a proper interaction with the transposition operator on P:

- For all $p \in P$ and $\langle\langle f_i, v_i\rangle\rangle_i$: $p(\langle\langle f_i, v_i\rangle\rangle_i) \Leftrightarrow p^T(\langle\langle 1 - f_i, v_i\rangle\rangle_i)$

In our running example of a protocol as a pair of types $\langle t_0, t_1\rangle$, we would define $\langle t_0, t_1\rangle$ to be true of a sequence $\langle\langle f_i, v_i\rangle\rangle_i$ if v_i has type t_{f_i}, for all i. The point of saying that an element of P is a predicate on a sequence of seen values is that it can thus be used to specify the communication pattern that must be obeyed along the arc, not merely the static properties of individual values.

We shall use the convention that $\tau_{u,a}$ denotes $\tau_{d,i}$, where d is the label of node u, and a is the i^{th} arc incident upon u. Similarly for $\sigma_{u,a}$.

- Given a state of a transaction graph, a *subsequent state* is one which differs at exactly one node u, where the new state at u is an element of:

  $$o_u(s_u) \cup \bigcup_a \tau_{u,a}(s_u, \sigma_{u/a,a}(s_{u/a}))$$

  Here, s_u and $s_{u/a}$ are states of nodes from the given graph state, and a ranges over arcs incident upon u.

Finally, an execution is a sequence of graph states, in which:

- Each state but the first is subsequent to the previous element in the sequence.
- On any arc, the values exposed along that arc satisfy the protocol on the arc.

Note that this definition implies that changes at nodes occur serializably—execution does not allow the two nodes on an arc to change simultaneously, i.e., two new exposed values cannot be based on two old exposed values.

We promised earlier some justification for the role of non-determinism. The first and foremost reason is that non-determinism is part of the reality that transaction graphs are designed to help cope with. Especially because of the modeling and summarizing role of applications, there is a necessity that the underlying formalism address non-determinism in a fundamental way. We will also see various technical conveniences of this formalization, such as the following: how do we know when execution of a transaction graph is finished? After all, the computational model here is quite distributed (as promised): there is no obvious "exit node", and specifying one, and its behavior, would be artificial, beneath the level of the rest of the above formulation. Rather, the fact that τ and o specify a *set* of states allows for a natural, *distributed*, termination condition.

$$\emptyset = \bigcup_u \Big(o_u(s_u) \cup \bigcup_a \tau_{u,a}(s_u, \sigma_{u/a,a}(s_{u/a}))\Big)$$

Obviously, an execution sequence stops once this condition holds. We shall see other technical conveniences of non-determinism in later sections.

Even though this paper does not directly address implementation concerns, we should note that an implementation of τ does not actually produce a set of states; rather, it eventually picks one state that is a member of the set produced by the theoretical τ. Thus, the definition of τ serves as a specification for the implementation rather than a prescription. (However, it may well be useful to implement the predicates $\tau_{d,i}(s, v) = \emptyset$ and $o_d(s) = \emptyset$.) Similarly, we do not propose literally implementing the predicate corresponding to an element of P. This too serves as a specification.

In the previous section, we saw that an activity description had a signature, and that there was a transposition operator on the protocols in signatures. These were necessary to specify well-formedness of transaction graphs. In this section, we added the following semantic notions, necessary in the definition of execution.

- For activity description d, whose signature is $(n_i : p_i)_i$, there are also the following components.
  - A set of states $\mathcal{S}_d$.
  - A list of maps $\sigma_{d,i} : \mathcal{S}_d \to V$ revealing part of a state to a neighbor.
  - A list of maps $\tau_{d,i} : \mathcal{S}_d \times V \to 2^{\mathcal{S}_d}$, corresponding to transactions along an arc.
  - A map $o_d : \mathcal{S}_d \to 2^{\mathcal{S}_d}$, corresponding to out-of-graph state changes.
- Each element of P acts as a predicate on values seen along an arc. The action as a predicate is well-behaved with respect to transposition on P.

We close this section by emphasizing the distributedness of the computational model, i.e., its suitability to a truly distributed implementation. There is no assumption of scheduling here, only that two adjacent nodes racing to perform a transaction on their common arc can decide who goes first. It may seem odd that when concurrency is the goal, transactions around a node must occur one at a time, but this is only the way that one says formally that transactions are serializ*able*, not that the implementation is required to perform them serially. Moreover, this approach to the formalism has the advantage of minimizing the machinery in the system, and maximizing the flexibility one has in making extensions, i.e., implementing functions specified by σ, τ, and o. This extends even to concepts that are often built into the semantics of a distributed system, for example, fairness in scheduling.

2.3 Graphical Condensation

In this section, we will show that there is a natural way to define an activity description that mimics the behavior of a transaction graph. This technique of shifting complexity between the graph structure and the activity descriptions has several important consequences. Most obviously, it means that transaction graphs can be specified hierarchically—one can place a node in a graph, much in the same manner that one writes a subroutine and refers to it by name.

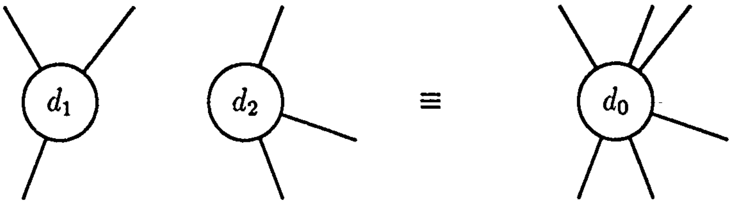
Less obviously, graphical condensation is related to obtaining different views of an activity, in the following way. Given a subgraph of a graph, that subgraph may be collapsed to a node as a way of summarization, in other words, a particular view of an activity may involve a particular condensation of certain aspects of the graph structure. A second view may condense a different subgraph, perhaps partially overlapping the first view. Thus the same activity may be viewed as occurring on quite different, not even hierarchically related, graphs.

To be precise, we will define the *induced activity description* for a transaction graph from the components of the activity descriptions on the nodes of the transaction graph, and of course, from the connectivity. Consider two graphs, one with a node labeled with the activity description induced from a graph, and one with the node replaced by the graph from which it was induced. The result will be that the execution of the two graphs are in 1-1 correspondence.

The following subsections will discuss graph condensation, in each case giving induced activity descriptions and a proof of the 1-1 correspondence of executions. There are two transformations, which together are sufficient to condense a graph to a node.

2.3.1 Pinching Two Nodes

In this subsection we consider the induced activity description for the graphical operation of "pinching" two nodes, i.e., replacing them with a single node, and attaching arcs that used to be incident upon either node to the new node. In its simplest form, the graph transformation is:



On the left is a transaction graph consisting of two unconnected nodes; on the right is a transaction graph consisting of a single node, having arcs corresponding to the arcs on the left, i.e., labeled in the same way (with graph-relative and node-relative names).

We first consider the signature for the new activity description d_0. Let $(n_{ji_j} : p_{ji_j})_{i_j}$, be the signature for d_j. We will assume that the sets of names are distinct, i.e., $n_{1i_1} \neq n_{2i_2}$, for any i_1 and i_2; if not, they may be renamed. Since we are trying to arrange that both graphs appear the same on the outside, they at least have to have the same signature, which forces the signature of d_0 to be defined as:

$$(n_{11} : p_{11}, \ldots, n_{1k_1} : p_{1k_1}, n_{21} : p_{21}, \ldots, n_{2k_2} : p_{2k_2})$$

Next, consider the set of states. If the two graphs behave the same, then in general we would have to have to define $\mathcal{S}_{d_0}$ to be $\mathcal{S}_{d_1} \times \mathcal{S}_{d_2}$. Similarly, identical behavior requires:

- $\sigma_{d_0,i}(\langle s_1, s_2\rangle) \overset{\text{def}}{=} \begin{cases} \sigma_{d_1,i}(s_1) & \text{if } i \leq k_1 \\ \sigma_{d_2,i-k_1}(s_2) & \text{otherwise} \end{cases}$
- $\tau_{d_0,i}(\langle s_1, s_2\rangle, v) \overset{\text{def}}{=} \begin{cases} \tau_{d_1,i}(s_1, v) \times \{s_2\} & \text{if } i \leq k_1 \\ \{s_1\} \times \tau_{d_2,i-k_1}(s_2, v) & \text{otherwise} \end{cases}$
- $o_{d_0}(\langle s_1, s_2\rangle) \overset{\text{def}}{=} o_{d_1}(s_1) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2)$.

This effectively defines the desired activity description d_0, which we will call $d_1 \times d_2$.

Result 1 Suppose we have two nodes u_1 and u_2 of a transaction graph, with activity descriptions d_i, and that we obtain a new transaction graph by pinching them into a single node u_0 which we label with $d_1 \times d_2$. Then there is a 1-1 correspondence between executions in the two graphs.

Proof The "correspondence between executions" can be made precise only by first being precise about the correspondence between states. In the two graphs, the states are given by the following respective sets:

$$\bigtimes_{u} \mathcal{S}_u \text{ and } (\bigtimes_{u \neq u_1, u_2} \mathcal{S}_u) \times \mathcal{S}_{u_0}$$

But $\mathcal{S}_{u_0} = \mathcal{S}_{u_1} \times \mathcal{S}_{u_2}$, so there is a clear correspondence between states, which involves only re-ordering and restructuring tuples.

$$\langle \ldots s_1 \ldots s_2 \ldots \rangle \leftrightarrow \langle \ldots \quad \ldots \quad \ldots \langle s_1, s_2 \rangle \rangle$$

With this correspondence of states, it is clear from the definition of σ_{u_0} and the fact that σ_{u_i} is unchanged for $i \neq 1$ or 2, that at corresponding states, the same values are exposed at the corresponding ends of all arcs.

The key to the argument is to look at the possible subsequent states of corresponding states. For any subsequent map in the "before" graph which differs at a node other than u_1 or u_2, there is trivially a subsequent map in the "after" graph differing at a node other than u_0, and vice versa. If the change occurs at u_i, changing s_i to s_i', there will be a subsequent map for the "after" graph in which $\langle s_1, s_2 \rangle$ is changed to $\langle s_1', s_2 \rangle$ or $\langle s_1, s_2' \rangle$. Conversely, any change at u_0 will be of this form, and thus there will be a corresponding subsequent state in the "before" graph. In short, there is a 1-1 correspondence between subsequent states, and hence a 1-1 correspondence between executions.

□

Define two activity descriptions to be equivalent, denoted $\cong$, if they have the same signature, if there is a 1-1 correspondence between states, and if under this correspondence, τ, σ and o are all equivalent.

Result 2 $d_1 \times d_2 \cong d_2 \times d_1$ and $(d_1 \times d_2) \times d_3 \cong d_1 \times (d_2 \times d_3)$

Proof In the first case, the correspondence between states is that between $\mathcal{S}_{d_1} \times \mathcal{S}_{d_2}$ and $\mathcal{S}_{d_2} \times \mathcal{S}_{d_1}$, the details of the proof are not worth writing. In the second, we use associativity of cartesian products to get correspondence of states, and give the proof for o.

$$
\begin{aligned}
& o_{(d_1 \times d_2) \times d_3}(\langle\langle s_1, s_2\rangle, s_3\rangle) \\
= \; & o_{d_1 \times d_2}(\langle s_1, s_2\rangle) \times \{s_3\} \cup \{\langle s_1, s_2\rangle\} \times o_{d_3}(s_3) \\
= \; & (o_{d_1}(s_1) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2)) \times \{s_3\} \cup \{\langle s_1, s_2\rangle\} \times o_{d_3}(s_3) \\
\cong \; & o_{d_1}(s_1) \times \{s_2\} \times \{s_3\} \cup \{s_1\} \times o_{d_2}(s_2) \times \{s_3\} \cup \{s_1\} \times \{s_2\} \times o_{d_3}(s_3)
\end{aligned}
$$

Starting from $o_{d_1 \times (d_2 \times d_3)}$, we get the same expression. Details for σ and τ are omitted.

□

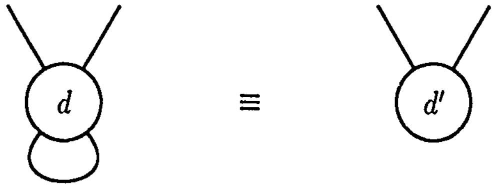
To summarize this section, we have shown that given any transaction graph G, we can construct a graph with a single node whose execution is essentially the same as that of G. The activity description for this node is given by:

$$\mathop{\times}_{u \in G} d_u, \text{ where } d_u \text{ denotes the activity description that } G \text{ assigns to } u$$

By the preceding result, the order in this product doesn't matter.

2.3.2 Shrinking Loops

While the techniques of the previous section condense an arbitrary transaction graph to one having only a single node, the activity description for that node is not what we want to call "the" activity description for the graph, because the pinching process leaves arcs both of whose ends are incident upon the single node; such arcs are called *loops.* Because pinching nodes does not affect the set of arcs, there will in fact be one loop on the single node for every non-dangling arc in the original graph. In this section, we consider the graph transformation of removing a loop.



The reason for the terminology "shrinking" is that while from the standpoint of only the graph structure, it looks as if the loop is simply being removed, from the point of view of the activity descriptions, the semantics for the loop is being pulled inside the node, i.e., turned into an out-of-graph change of state.

Let d have signature $(n_i : p_i)_i$. For convenience, we will assume that the loop has node-relative names n_1 and n_2, so that the signature for d' is $(n_i : p_i)_{i>2}$. We naturally define $\mathcal{S}_{d'}$ to be $\mathcal{S}_d$, and carry over the definitions of $\sigma_{d,i}$ and $\tau_{d,i}$ for $i > 2$. The only non-trivial part of the construction is that what were previously transactions along the loop must become out-of-graph changes, as seen in:

- $o_{d'}(s) \overset{\text{def}}{=} o_d(s) \cup \tau_{d,1}(s, \sigma_{d,2}(s)) \cup \tau_{d,2}(s, \sigma_{s,1}(s))$

This completes the definition of d', which we denote by $d - [1,2]$; more generally we give the pair of indices removed. We will also use the notation $d - a$ when it is understood that the indices arise from an arc a that is a loop at a node labeled by d.

Result 3 Suppose a node u of a transaction graph is labeled by d and has a loop a, and that we obtain a new transaction graph by removing arc a at u and replacing the label with $d - a$. Then there is a 1-1 correspondence between executions in the two graphs.

Proof Unlike the case in the previous subsection, here the sets of states are identical. For subsequent states that differ at a node other than u, the correspondence is immediate, as is the case of u when the state change is due to a transaction along an arc other then a. The remaining state changes at u in the "before" graph are either out-of-graph changes, or transactions along a, all of which correspond to out-of-graph changes at the corresponding node of the "after" graph, and conversely. Thus there is a 1-1 correspondence in subsequent states.
□

Not only does the order of shrinking arcs not matter, but the operation "commutes" with pinching:

Result 4 Let i_1, i_2, i_3 and i_4 be distinct indices of the signature of an activity description d. Then:

$$(d - [i_1, i_2]) - [i_3, i_4] \cong (d - [i_3, i_4]) - [i_1, i_2]$$

Result 5 Let i_1, i_2 be distinct indices of d_1, and let d_2 be an activity description. Then:

$$(d_1 - [i_1, i_2]) \times d_2 \cong (d_1 \times d_2) - [i_1, i_2]$$

(The notation here assumes that signature indices of d_1 correspond to indices for the "d_1 part" of $d_1 \times d_2$.)

Proof In both cases, the signatures are the same, as are the set of states and the actions of σ and τ. In the first result, the expression for o for the left hand activity description expands out to:

$$\left(o_d(s) \cup \tau_{d,i_1}(s, \sigma_{d,i_2}(s)) \cup \tau_{d,i_2}(s, \sigma_{d,i_1}(s))\right) \cup \tau_{d,i_3}(s, \sigma_{d,i_4}(s)) \cup \tau_{d,i_4}(s, \sigma_{d,i_3}(s))$$

The right hand activity description expands out to a similar expression with i_1 and i_2 exchanged with i_3 and i_4 respectively. So the result follows by commutativity and associativity of "$\cup$".

In the second result, the computation is messier, but for completeness, here it is:

$$\begin{aligned}
& o_{(d_1 - [i_1, i_2]) \times d_2}(\langle s_1, s_2 \rangle) \\
= \; & o_{d_1 - [i_1, i_2]}(s_1) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2) \\
= \; & \left(o_{d_1}(s_1) \cup \tau_{d_1,i_1}(s_1, \sigma_{d_1,i_2}(s_1)) \cup \tau_{d_1,i_2}(s_1, \sigma_{d_1,i_1}(s_1))\right) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2) \\
= \; & o_{d_1}(s_1) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2) \cup \tau_{d_1,i_1}(s_1, \sigma_{d_1,i_2}(s_1)) \times \{s_2\} \cup \tau_{d_1,i_2}(s_1, \sigma_{d_1,i_1}(s_1)) \times \{s_2\} \\
\cong \; & o_{d_1 \times d_2}(\langle s_1, s_2 \rangle) \cup \tau_{d_1 \times d_2, i_1}(\langle s_1, s_2 \rangle, \sigma_{d_1 \times d_2, i_2}(\langle s_1, s_2 \rangle)) \cup \tau_{d_1 \times d_2, i_2}(\langle s_1, s_2 \rangle, \sigma_{d_1 \times d_2, i_1}(\langle s_1, s_2 \rangle)) \\
= \; & o_{d_1 \times d_2 - [i_1, i_2]}(\langle s_1, s_2 \rangle)
\end{aligned}$$

In the "$\cong$" step, we are using the fact that since i_1 and i_2 are indices for d_1, $\tau_{d_1,i}(s_1, v) \times \{s_2\} \cong \tau_{d_1 \times d_2, i}(\langle s_1, s_2 \rangle, v)$, for $i = i_1$ and i_2, and similarly for $\sigma_{d_1,i}$.
□

From the first of these results, we can use the notation $d - \{[i_j, i'_j]\}_j$ to mean $d - [i_1, i'_1] - [i_2, i'_2] - \ldots$, because the order doesn't matter. Similarly, if $\{a_i\}_i$ is a set of arcs, we can use the notation $d - \{a_i\}_i$ without ambiguity.

With these results, it is possible to condense an arbitrary transaction graph G to a graph with a single node and only dangling arcs, and having essentially the same execution as G. The activity description on the single node, which we call the *activity description induced by G*, is given by:

$$d_G \stackrel{\text{def}}{=} \bigtimes_{u \in G} d_u - \{a \in G \mid a \text{ is non-dangling}\}$$

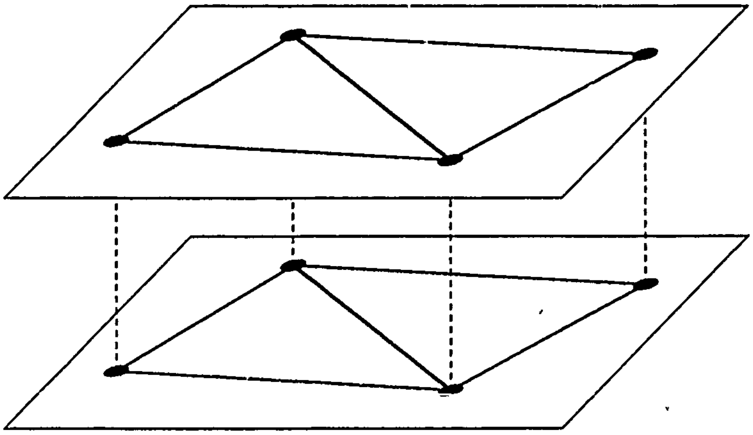
In many respects, the ability to obtain an activity description that corresponds to a graph dictated the technical details of the definition of an activity description. For example, suppose that we had defined an activity description to allow simultaneous change of exposed values. Then there would be a class of activity descriptions which could not be realized as transaction graphs. Or suppose that we had not allowed out-of-graph changes to the state. Then transaction graphs would not be closed under the shrinking of loops.

In summary of these first two subsections, we have seen that the definition of activity descriptions and transaction graphs is done in such a way that the complexity of a transaction graph can be traded off against the complexity of the state of its nodes. There is nothing about the behavior of an activity description that is particularly different from the behavior of a transaction graph; in other words, there is no particular aspect of behavior that *must* be put into a graph, or must *not* be put into a graph. We believe that this property enables the construction of a layered system of activity descriptions with clean encapsulation.

2.4 Projected Execution

In viewing an ongoing activity, it is important to be able to select only part of the information that is necessary for the execution of the transaction graph as a whole. Naturally, which information is selected depends upon the view in question. The goal of this section is to structure the selection of information in such a way that a view makes sense on its own. The technical approach is to require that the state changes seen in the view are actually an execution in a transaction graph that is related in some well-understood way to the transaction graph being viewed. Thus, where in the previous section we studied transformations with 1-1 correspondence in executions, here we are interested in transformations which have the property that for every execution in the original transaction graph (the one being viewed), there is an execution in the transformed transaction graph (the view), but not necessarily vice-versa. That is, the set of executions in the view contains (and may greatly exceed) those in the original graph. This is because of the loss of information in the transformed graph, which causes an increase in the non-determinacy of σ, τ and o.

In further contrast to the previous section, there we always changed the graph structure, but kept 1-1 correspondence in the states. Here, it suffices to keep the graph structure the same, and change only the activity descriptions on nodes. Imagine two copies of the same graph structure (not transaction graphs, yet), one drawn in a plane directly above the other.



Turn each of these graphs into a transaction graph by assigning an activity description to each node. Further, assign a state to each node, where the states on each of the planes together make up the state of an execution going on in that plane. The bottom plane we will view as the detailed execution, and the top plane as a view on the detailed execution—a "filter" through which we look at the "real" execution. This view is formalized as a projection mapping the bottom to the top plane, where this projection behaves "properly" (one might say homomorphically) with respect to the functions σ, τ, and o.

The essence of the subject here is maps from one activity description to another. We will denote such maps with Π, and by abuse of notation, also use Π for the map of constituent parts of an activity description. A map Π must have the following constituents:

- A map that takes signatures to signatures: $(n_i : p_i)_i \mapsto (n_i : \Pi_i(p_i))_i$
- A map that takes a state to a state: $\Pi : \mathcal{S}_d \rightarrow \mathcal{S}_{\Pi(d)}$
- Maps for values seen along an arc: $v \mapsto \Pi_i(v)$
- Maps (functionals) taking $\sigma_{d,i} \mapsto \sigma_{\Pi(d),i}, \tau_{d,i} \mapsto \tau_{\Pi(d),i}$, and $o_d \mapsto o_{\Pi(d)}$

We are interested only in the subset of such maps, for which we use the term *projections*, that have the properties given below. In stating these properties, we further abuse Π by applying it to subsets of $\mathcal{S}_d$, by which we mean the subset of $\mathcal{S}_{\Pi(d)}$ obtained by applying Π to elements of the given subset.

- For all $s \in \mathcal{S}_d$, $\Pi_i(\sigma_{d,i}(s)) = \sigma_{\Pi(d),i}(\Pi(s))$
- For all $s \in \mathcal{S}_d, v \in V$: $\Pi(\tau_{d,i}(s,v)) \subseteq \tau_{\Pi(d),i}(\Pi(s), \Pi_i(v))$
- For all $s \in \mathcal{S}_d$: $\Pi(o_d(v)) \subseteq o_{\Pi(d)}(\Pi(s))$

The "$\subseteq$" in each of the last two conditions make precise the ways in which $\Pi(d)$ may lose information present in d.

In addition to homomorphic behavior at each node, we need a similar condition for what happens on an arc:

- Let Π be a projection of signatures, and let Π_0 and Π_1 be projections of values. We say that Π_0 and Π_1 are *consistent with* Π *at* p $\overset{\text{def}}{\Leftrightarrow}$ for any sequence of values $\langle\langle f_j, v_j \rangle\rangle_j$:

  $$p(\langle\langle f_j, v_j \rangle\rangle_j) \Rightarrow \Pi(p)(\langle\langle f_j, \Pi_{f_j}(v_j) \rangle\rangle_j)$$

  In other words, Π_0 and Π_1 project a legal sequence of values to a legal sequence in the view.

This property is used in the definition of a projection of a transaction graph G, which is a set of projections Π_u, one for each node of G, with the properties:

- Replacing the label d_u on node u with $\Pi_u(d_u)$, for all $u \in G$, results in a transaction graph, i.e., one in which the rule for signatures is met. (In other words, if u_1 and u_2 are connected by arc a, and if u_1 has protocol p_1 on a (so $p_1^T = p_2$), then $\Pi_{u_1,a}(p_1^T) = \Pi_{u_2,a}(p_2)$.)
- For every arc, the projections of the values at the ends of the arc are consistent with the projection of the protocol on the arc.

Again overloading Π, let $\Pi(G)$ be a projection of a transaction graph G, and for any state $\{s_u\}_{u \in G}$ for G:

- Let $\Pi(\{s_u\}_{u\in G}) \overset{\text{def}}{=} \{\Pi_u(s_u)\}_{u \in G}$

Finally, we need to define what it means to project an execution of a graph. A naive definition would be that a projected execution is obtained by applying Π to each state in the execution sequence. This is almost the appropriate definition, but overlooks the situation in which consecutive states of the detailed execution project to the same state—a possibility we certainly want to allow, so that projections can reduce the number of steps in an execution as well as summarize state.

- Let $\langle s_i \rangle_i$ be a sequence of states. We define $\Pi(\langle s_i \rangle_i)$ to be the sequence $\langle \Pi(s_{i_j}) \rangle_j$ where $\langle i_j \rangle_j$ is defined by:

$$i_1 = 1 \text{ and } i_{j+1} = \text{ the smallest } i > i_j \text{ such that } \Pi(s_i) \neq \Pi(s_{i_j})$$

This completes the set of definitions we need for the following result:

Result 6 Let G be a transaction graph, and $\Pi(G)$ a projection. Then Π projects any execution in G into an execution in $\Pi(G)$.

Proof Let $\langle s_i \rangle_i$ be an execution in G, and $\langle s_{i_j} \rangle_j$ its projection. We must prove that for $j > 1$, $\Pi(s_{i_j})$ is a subsequent state to $\Pi(s_{i_{j-1}})$. In G, we know that s_{i_j} is a subsequent state to $s_{i_{j-1}}$, that the state change was localized at a node, and caused either by τ or by o. Let s_u be the state of u at $s_{i_{j-1}}$. Then if the change was caused by τ:

$$\begin{aligned} \Pi_u(\tau_u(s_u, \sigma_{u/a,a}(s_{u/a}))) &\subseteq \tau_{\Pi(a),a}(\Pi_u(s_u), \Pi_{u/a,a}(\sigma_{u/a,a}(s_{u/a}))) \\ &= \tau_{\Pi(a),a}(\Pi_u(s_u), \sigma_{\Pi(u/a),a}(\Pi_{u/a,a}(s_{u/a}))) \end{aligned}$$

By $\tau_{\Pi(u)}$, we of course mean $\tau_{\Pi_u(d_u)}$, where d_u is the activity description that G places on node u. Thus, $\Pi(s_{i_j})$ is subsequent to $\Pi(s_{i_{j-1}})$. The case for o is similar.

The other requirement for an execution is that the values along an arc satisfy the protocol. This is true for the sequence of states seen in $\Pi(G)$ because it holds for these in G, and by the consistency requirement on the projections of values.

□

The point is that whatever can be ascertained about executions in the view can be understood as a result about the detailed execution. Even when using the view as an inspection device, the condition that it is a projection means that the viewer can understand what is and what might happen next in a self-contained way—specifically, in terms of σ, τ, and o in the projections.

2.5 Summary

We have defined a graph-based scheme for the computat.,onal tracking and modeling of activity. The basic building block is an *activity description*, which describes "local" activity. An activity description governs how its instances can be connected to other instances of activity descriptions by specifying a *protocol* for each neighbor. It also governs the evolution of the state of an instance of the activity description: the parts of the state that are revealed to neighbors, the rules for changing that part of a state, and the rule for changing the part of the state not seen by any neighbor.

Structurally, a *transaction graph* is a graph whose nodes are labeled with activity descriptions. Arcs between nodes receive two protocols, one from the activity description on the node at each end. There is a conformance rule requiring that the two protocols be the same, up to a symmetry operation. Transaction graphs are allowed to have dangling arcs, each of which receives only one protocol. The set of protocols given by dangling arcs turns out to play the same role as the set of protocols specified by an activity description.

We defined the notion of an execution of transaction graphs. This consists of execution steps at each of the nodes, governed by the activity description on a node. At any step, a node may change its state independently of its neighbors, or it may transact with a neighbor: it may change its state in a way that depends upon its state and the part of the neighbor's state made visible to it. (The terminology "transaction" comes from the fact that such steps are required to be serializable: adjacent nodes may race to perform a transaction along an arc, but one or the other will go first.) The sequence of values exposed along an arc must obey the protocol on the arc.

It is important that the definition of execution not require any central component in an implementation. We wish to support applications in which the nodes represent loosely dependent activities, and in which the activities may be proceeding at distant sites. Thus the formalism makes quite explicit exactly where the communication happens at each stage, and it requires only point-to-point communication, so there is no need for a global communication or locking mechanism.

Much of the discussion here has been toward the development of a calculus of operations on transaction graphs. The first part of this discussion showed how to preserve the execution under a set of natural graph-theoretic transformations, while the second part left the graph structure invariant and showed how to characterize summarization of execution.

The reader may be concerned that transaction graphs as a formalism do not provide an immediate basis for the construction of activity coordination programs by novices. In fact, there is no claim to the contrary, and as we stated at the outset, our goal here is to provide the intellectual basis for an activity coordination system. This basis must meet other criteria: it must provide a coherent "mentality" for an eventual system, by providing concepts that are applicable in its seemingly disparate parts. In chapter 3, we shall see how the somewhat theoretical ideas developed here relate to real world aspects of an activity coordination system, and in chapter 4 we develop several higher level activity descriptions and operations on transaction graphs.

3 Connections to Reality

3.1 Parameterization and Instantiation

One never specifies an activity description directly, because of the necessity to parameterize its associated functions and even its set of states. We thus introduce the notion of a *parameterized activity description.* This is a function which, when applied to arguments, yields an *activity instantiation*, consisting of the following components.

- A signature.
- A list of functions corresponding to σ_i.
- A list of functions corresponding to τ_i.
- A function corresponding to o.
- A *state.*

For example, we don't often want to specify an activity description with a particular person wired in, but rather a parameterized activity description which can be invoked with a person to be named later. Naturally, the parameterization extends to transaction graphs, which have the additional point that the same parameter may be referenced by several nodes.

Recall that in section 1 we spoke of the need for extensions both at a primitive level and at a graphical level. In an implementation, there would be a way to write parameterized activity descriptions in whatever language was appropriate (e.g. C, PASCAL, or Lisp); this would satisfy the need to extend the set of primitives. An implementation would also provide a means for constructing parameterized activity descriptions in terms of its graphical structure, by specifying:

- A header for the formal parameters of the activity description (not to be confused with the names on dangling arcs, which become part of the signature of the activity description).
- A graph in which each end of an arc is labeled with a name (at least implicitly; conventions can be used to reduce the number of names that must actually be given directly).
- For each node in the graph, an expression which will evaluate to an activity instantiation whose signature has names corresponding to the nearby names on arcs incident upon this node. These expressions typically involve the above-mentioned formal parameters.

When an activity description is specified in this way, its application to actual parameters yields an activity instantiation whose signature, σ, τ, o, and state are induced, as described in section 2.3, from the activity instantiations that result from evaluating the expressions on the nodes. This provides a more intuitive level at which to construct activity descriptions.

We have thus far mentioned two extremes in specifying parameterized activity descriptions: the primitive level, specified entirely in a standard programming language, in which the resulting activity instantiation has no internal graphical aspect, and the "high" level, specified in terms of a static graph, and thus having the graphical decomposition. An implementation should also provide for hybrids of these two extremes: it would be useful to be able to write parameterized activity descriptions which produced activity instantiations whose graph structure depended upon the parameters, for example, the number of arcs leaving a node might depend upon the size of a set parameter. The resulting instantiation would appear to a user as if it were executing a graph-based activity description, rather than a primitive activity description. Thus, hybrid parameterized activity descriptions provide a powerful means of encapsulating regular graph structure, even if it is not statically fixed by a parameterized activity description.

3.2 Operations on Activity Instantiations

In the previous section we defined the fundamental means of constructing an activity instantiation, namely, the application of a parameterized activity description to actual parameters. In this section, we discuss operations on these objects. The most obvious are to set one in motion—to begin execution—and to view the current state of the state. Indeed, these will probably be the principal operations done by participants in coordinated activities.

Equally important to implementers and extenders of the system are implemented versions of some of the operations on transaction graphs and node descriptions discussed in chapter 2. In particular, it is possible to obtain activity instantiations not only by their ab initio construction using parameterized activity descriptions and transaction graphs, but also by operations that have activity instantiations as arguments. Corresponding to the Results of section 2.3 and 2.4, the following are candidates for implementation:

- pinching two vertices—takes an activity instantiation based on a graph, and a pair of nodes of that graph; yields an activity instantiation based on a graph obtained as described in Result 1.
- shrinking loops—similar to the above, but based on Result 3.
- useful combinations of the above, for example an operation that takes an activity instantiation based on a graph and a set of nodes of that graph, and yields an activity instantiation based on a graph obtained by pinching all the nodes in that set and shrinking all the loops incident upon the new node.
- projection—takes an activity instantiation and a rule for projecting its signature, functions, and state; yields an activity instantiation as described in Result 6. (If the given activity instantiation is based on a graph, the projection rule would be specified node-by-node.)

It is important that all these operations be done in such a way that the original instantiation and the resulting instantiation conceptually share the same state, so that execution of one is quite directly an execution of the other.

Another essential operation on activity instantiations, one that has no interesting mathematical counterpart, is this:

- copy—takes an activity instantiation and yields a distinct activity instantiation with the identical state.

This has obvious essential roles in debugging and backup, as well as uses discussed in later sections.

While there is no claim that the above list of operations on activity instantiations is exhaustive, and while we would certainly consider extending it, we nevertheless claim, on the basis of later sections of this chapter and of chapter 4, that the operations have wide application.

3.3 User Interface

A crucial aspect of an activity coordination system is its user interface. In this section we discover that the program that deals most directly with the user—displaying the state of activity instantiations and passing directives from the user to activity instantiations—can literally be an activity description in the process of executing. Thus, there is no fundamental distinction between a user's connection to the system and the way that the system works internally.

We will begin by discussing a particular projection of an activity instantiation (as discussed in the previous section), called *interface*. The result of *interface* is an activity instantiation whose activity description has a degenerate τ and o—mathematically, $\tau(s,v) \overset{\text{def}}{=} 2^S$ and $o(s) \overset{\text{def}}{=} 2^S$ for every s and v, i.e., from the point of view of τ and o, the next state is completely unspecified. In the implementation, the meaning of this is that all the changes to the state arise from out-of-graph communication.

While τ is totally useless in constraining behavior, it is quite useful in other respects. To explain how, it is first necessary to describe the state of the projected activity instantiation. There are two aspects to it:

- It shares state with the argument of *interface.*
- It contains information that indicates how to display the result.

The fact that the result of *interface* shares state with the argument means that it can display it on the user's screen. The additional information records the layout of nodes (which the user may change), scrolling position, and so on.

In the other direction, the result of *interface* also is aware of what the user is doing—typing, mousing, clicking—and is responsible for translating these into the form of out-of-graph messages expected by the argument to *interface.*

The above discussion has talked about an *interface* projection as if there would be only one. Even if this were the case, there is an advantage to implementing it as a projection, because it is not necessary to design into the basic machinery any "special features" which enable communication with a user. A larger payoff is that in reality there will be several interfaces. For example, in the development of the system, we will probably first want a simple text interface, and later, a more graphical one. Another example is an interface that would tie activity instantiations to an active database. Yet another example is considered in the next section.

3.4 History

In the language of transaction graphs, a history is equivalent to an execution—it is simply a sequence of states. In an implementation, a history is obtained by interfacing a journal keeper to an activity instantiation. In other words, a history interface responds to a change not by updating the screen, but appending an entry to a file.

Not just any entry will do, of course. The point of a history is to be able to reconstruct an execution, which constrains the nature of the appended entry. The natural way for a user to view history is with exactly the same machinery used to view the original execution, except that (a) it will not be possible to affect the execution (e.g., by making a decision), and (b), it will be possible to go back and forth in time, either by steps, or to the state at a particular time. This specification suggests a natural implementation, in which literally the same interface is set up with a state that is shared not with the state of the original activity instantiation, but with the state of a "history viewer", itself an activity instantiation. The arguments to the parameterized activity description yielding a history viewer are the history interface instantiation and the journal file. It begins by copying the state and noting the corresponding point in the journal file. Its execution consists of listening to the user say things like "go forward", "go back", and "go to time x"; the changes will automatically appear in interfaces referring to the history viewer.

A pragmatic difficulty in dealing with history is how much of it to keep. It is fair to say that this is not a system problem, because it is ultimately up to the user, but it is also true that the system must provide facilities that allow the user to cope with the problem. The most obvious requirement is that it must be possible to cut off the front end of a journal; events that happened too long ago (or that are now archived off-line) can be forgotten. The more interesting issue is the amount of detail that is kept, a problem already mentioned in connection with observing an ongoing activity. Naturally, we propose the same means to deal with it, namely, projection. Thus, while a history is always a recorded execution, that execution may not be the lowest level execution, but rather a projection of it that removes extraneous detail.

The fact that history recording and viewing are implemented as activity descriptions, and the use of projection to remove detail, makes possible useful combinations of these techniques. For example, one might record quite detailed history, which is kept on the disk for only a limited period of time. As the detailed history is deleted, (and possibly archived) it can be projected onto a less detailed history, which is small enough to keep over longer stretches. Note that a projection may be designed after the recording of detailed history, so if at some later phase in a project, one wants to see a certain projection of history, one goes to the archive and re-executes the history, projecting it in the desired way.

3.5 Implementation of σ, τ, and o

In the formalization of activity descriptions, we defined τ_i and o as functions that yield a set of possible states, and an execution step as a choice from the union of the results of τ_i and o. In connecting transaction graphs to reality, the choice must be resolved, and to keep the implementation distributed, it is natural to impose the responsibility for the resolution on the implementation of an activity description. Thus, while the functions τ_i and o are formally non-deterministic, an implementation of an activity description is deterministic—what is non-deterministic in reality is the order in which events occur.[1]

An activity instantiation may need to persist for weeks or months, and as we have emphasized, may take place on a widely distributed network. Thus, while we tend to think of an activity instantiation as an execution, it cannot correspond literally to, for example, a Unix process. In [Kar89] we developed machinery for connecting a database and an operating system to provide a facility for persistent execution. This machinery provides a natural basis for an implementation of activity descriptions, as we now briefly outline.

Let us assume that we have an activity instantiation, based on a graph, and that all of its nodes are "asleep", i.e., stored on a persistent medium. In this state, the activity instantiation is waiting for an out-of-graph event, such as the passage of time or a user action. In this discussion we omit the details, but such an event has the effect of "awakening" a node of the graph, causing the execution of a "transition program" [Kar89]. The implementation of the activity description for the awakened node is in its transition program. The execution of the activity description takes whatever action is appropriate, including perhaps transactions with other nodes, which may awaken their transition programs (perhaps on distant computers). Eventually, the transition program for the awakened node completes its set of actions, and puts the activity instantiation for that node back to sleep. Other nodes, whether awakened by the node first awakened or by independent out-of-graph events, may still be awake, and go on executing their transition programs independently.

This brief outline necessarily omits many issues which would have to be considered in a full-fledged design for an implementation, but there are several points that are natural to raise even at this level of detail. First, we note that while the theoretical discussion is in terms of σ, τ, and o, an implementation of a primitive activity description is, at the outer level, a dispatch on the reason for which it is being awakened—some kind of out-of-graph event, or because of a transaction initiated by a neighbor. Second, whatever the reason a node is awakened, it may engage in transactions with its neighbors, when it does so, it must be prepared to cope with the rejection of its transaction, and allow the neighbor's transaction to occur first. This is necessary because in a distributed system communication takes time, and independent nodes may "decide" to engage in a transaction without knowing, until it is too late, that they are in a race. Third, when an activity instantiation for a node u is asleep, then $\tau_u(s_u, v_{u/a,a}) = \emptyset$ for all arcs incident upon u. If in addition $o(s_u) = \emptyset$, the vertex is said to be *stable*—it can be changed only by transactions. Thus, if the activity instantiations for all the nodes of the graph are stable, the termination condition of section 2.2 is met. This suggests that each activity instantiation record whether $o(s_u) = \emptyset$, thereby enabling

[1] This is a slight over-simplification—we would not wish to preclude the use of deliberately randomized algorithms in activity descriptions.

detection of termination.

Finally, there is the necessity to provide for shared state, as we discussed in previous sections. In a distributed system, it is not feasible to do this literally. However, the desired effect can be achieved by establishing the convention that implementation of an activity description announce changes to its state. Activity instantiations that view other activity instantiations arrange to receive out-of-graph events when such changes occur, and are thus able to track the progress of viewed activity instantiations. (We have used this technique successfully in the implementation of the E-L system.)

In summary, the implementation of primitive activity descriptions, in so far as their incorporation into the activity coordination system is concerned, is a well-structured task. These functions must:

- respond to possible events, including transactions of neighbors and out-of-graph events;
- when necessary, engage in transactions with neighbors, which may possibly be rejected;
- announce changes of state; and
- indicate whether the activity instantiation is stable.

We anticipate no great difficulties in filling out the design and doing the implementation.

3.6 Cutover

In section 1, we mentioned the "cutover" problem: how to change descriptions of activities while the activities themselves are under way. We are now in a position to pose the problem in a more formal way: how do we modify an activity instantiation whose execution is under way? We can further set forth a criterion for the form a solution should take:

- If both the old and new activity descriptions are specified as transaction graphs, we would like to express the cutover technique at the level of transaction graphs.

Recall that execution is defined for only those activity descriptions with no dangling arcs, i.e., with an empty signature (section 2.2). Let us consider an activity description d with no internal graph structure, or at least where the graph structure has been removed by condensation to a single node, as described in section 2.3, so that the only function to worry about is $o : S_d \rightarrow 2^{S_d}$. The most general definition of a *cutover* is a map that takes o, s to o', s', the idea being that execution continues using o' on s'. There are two problems with this definition. First, even given the description of the map, what are the engineering issues in actually effecting the cutover of an activity instantiation? Second, the definition is so general that it is useless--it provides no insight into how to obtain o' and s' in a reasonable way.

We consider first the engineering issue. The restrictions we have already imposed on activity instantiations make this problem less terrifyingly impossible than it might first appear. In particular, the fact that the state of an execution can be written to disk and restored to fast memory means that there is a well-specified clean point at which cutover can occur. In implementation terms, all that is necessary is to ensure that the activity instantiation is in persistent storage (rather than fast memory), and then to change the transition program and state associated with with it. All of the many issues that would arise in cutting over an executing UNIX process simply don't arise.

We now turn to the issue of producing o' and s'. The easy part is producing a new o': this corresponds to changing the program, an activity with which we are all familiar. As a simple first case, then, we characterize the situation in which $s = s'$ (but $o \neq o'$) as *state-invariant* cutover. In addition to being used in the obvious way, to indicate that from now on, things will be done differently, state-invariant cutover may be used to recover from states that are erroneous because of bugs in an activity description. Recall that the history viewer literally operates on a states. The technique is to back up in time until a satisfactory state is reached, and then to resume execution with the new activity description (o') and the desired state (s'). (This technique assumes that history is recorded at the level of detailed execution, making that a wise thing to do during debugging.)

We now drop the assumption that the interesting graph structure in an activity description has been condensed out, and propose an approach to cutover that exploits the fact that activity descriptions often arise as transaction graphs, composed by connecting simpler activity descriptions having non-empty signatures. The basic idea is to support cutover at a vertex of a transaction graph leaving the state and activity description at all other nodes unchanged.

One might claim that in general it is not enough to be able to change a single node; it might be necessary to restructure an entire subgraph. Indeed this is the case, *but* doing so actually involves three steps.

1. Restructure the graph by condensing and uncondensing subgraphs to isolate the change at a single node.

2. Effect a cutover at the node.

3. Restructure the graph to correspond to the desired structure of the revised activity description.

The restructuring of a graph is one that produces a new σ', τ', o' and s' jointly, but in a way which is very stylized and preserves the essential semantics of the program. As indicated, the new functions are produced from the old by graph operations. From section 2.3, it is evident that s' differs from s by transformations of the form $\langle \ldots \langle s_1, \ldots, s_k \rangle \ldots \rangle \leftrightarrow \langle \ldots, s_1, \ldots, s_k \ldots \rangle$, i.e., just by grouping operations on tuples. The operations on tuples can be carried out automatically, given the transformations that take τ to τ'.

There remains the problem: how does a person conveniently create a new state? The answer is, the same was as always, by executing an activity description. To be more explicit, suppose that in graph G, there is a node u which we wish to cut over to a new σ'_u, τ'_u, o'_u, and s'_u. Then, in general, we construct a G' with vertex u' whose activity description has the same signature as that on u, with $\sigma_{u'}, \tau_{u'}$, and $o_{u'}$ being the desired σ'_u, τ'_u, and o'_u. We then execute G' to produce the desired s'_u as the state at u'. The final step in the cutover is to say that node u in the activity instantiation based on G is to be cut over to node u' in the instantiation based on G'. This, like state-invariant cutover and graph-restructuring, can be done in generic way. We should note that it also has the effect of deleting the state at u in the execution of G, and at all nodes in the execution of G' other than u'. (An interesting variation is to continue the execution G', but for its state at u' to become the state that was at u, in other words, for G and G' to exchange states between u and u'. Admittedly, no application springs immediately to mind.)

There is no claim here that this trivializes the problem of cutover. This design does, however, satisfy the criterion that cutover be accomplished at the same level as the changes to activity descriptions. In particular, a person who is working solely at the level of transaction graphs (i.e., not introducing primitive transaction protocols) can effect cutover solely in terms of transaction graphs.

4 Some Common Idioms

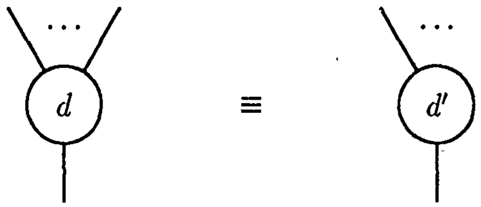
The transaction graph formalism was deliberately designed to favor formal simplicity and composability over the inclusion of many "features" at a fundamental level. The rationale is that if the basic design is clean and powerful, the desired features can be programmed and encapsulated. In the sections below, we consider common patterns of graph manipulation that can be done using the fundamental operations described in section 2.3, and we provide several paradigmatic activity descriptions. The aim here is not to be complete, but to demonstrate that a high-level system can indeed be build on this foundation.

4.1 Deleting Arcs and Nodes

The basic graph operations previously discussed provide the ability to unconditionally transform graph structure, yet to retain the semantics of execution. It is clearly not possible to unconditionally delete an arc or node and retain execution semantics, but there is an obvious condition under which this is possible. For arcs, a sufficient condition is that activity descriptions on the nodes at each end of the arc behave independently of the value seen along the arc. To be more precise, we say that *d ignores arc* $i_0 \overset{\text{def}}{\Leftrightarrow} \tau_{d,\imath_0}$ is independent of its second argument, i.e., there is a function $\theta_d : S_d \rightarrow 2^{S_d}$ such that:

- $\tau_{d,i_0}(s, v) = \theta_d(s)$, for all $s \in S_d$.

The fundamental transformation is on activity descriptions:



Let the activity description on the left be d with signature $(n_\imath : p_\imath)_\imath$ and assume d ignores i_0; for convenience, that on the right will be d' with signature $(n_\imath : p_i)_{\imath \neq i_0}$ and functions:

- $\sigma_{d',i}$ and $\tau_{d',i}$ are the same as $\sigma_{d,i}$ and $\tau_{d,i}$, respectively, for $i \neq i_0$.

- $o_{d'}(s) \overset{\text{def}}{=} o_d(s) \cup \theta_{'}(s)$

Suppose that a graph has nodes u_1 and u_2 joined by an arc a_0, and that each of u_1 and u_2 has an activity description that ignores a_0. The operations to delete the arc may be expressed as follows:

- Pinch the nodes at each end of the arc, and shrink a_0 (now a loop), obtaining the following functions for now node u_0:
  - $\sigma_{0,a}(\langle s_1, s_2 \rangle)$ for $a \neq a_0$ carries over from $\sigma_{\imath,a}(s_i)$, where a was incident upon u_i.
  - Similarly for $\tau_{0,a}(\langle s_1, s_2 \rangle, v)$.

- $$\begin{aligned} o(\langle s_1, s_2\rangle) &= o_1(s_1) \cup o_2(s_2) \cup \tau_{1,a_0}(s_1, \sigma_{2,a_0}(s_2)) \cup \tau_{2,a_0}(s_2, \sigma_{1,a_0}(s_1)) \\ &= o_1(s_1) \cup \theta_1(s_1) \cup o_2(s_2) \cup \theta_2(s_2) \end{aligned}$$

  The first equality comes from shrinking $a_{0,i}$ and the second, from the assumption that u_i ignores a_0, for $i = 1$ and 2.

- Unpinch u_0 to obtain u'_1 and u'_2 with activity descriptions $d'_j, j = 1$ and 2, where:

  - $\sigma_{d'_j,a}$ and $\tau_{d'_j,a}$ for $a \neq a_0$ are the same as $\sigma_{d_j,a}$ and $\tau_{d_j,a}$, respectively.
  - $o_{d'_j}(s) = o_{d_j}(s) \cup \theta_{d_j}(s)$

  Thus u'_j has the above-described transformation of d_j.

The results for pinching and shrinking guarantee that the graph with the deleted arc has executions in 1-1 correspondence with the original graph.

Why would anyone write a transaction graph with a deletable arc? One probably would not, at least not directly. The real utility of this transformation is in its combination with projection—even though an activity description does not ignore i_0, a projection might. Thus projection not only simplifies states and shortens execution sequences, it can also be viewed as simplifying the pattern of coordination. This is a formal property that corresponds to real-world experience: at a gross level, certain activities may be described as independent, while if one takes a closer look, it becomes clear that dependencies do exist.

For deleting a node, the condition is that its activity description has the following property:

d is *boring* $\overset{\text{def}}{\Leftrightarrow}$ its signature is empty and $o_d(s) = \emptyset$ for all s.

Let a node u_1 have a boring activity description. Then u_1 can be deleted as follows:

- Pinch u_1 together with any other node u_2, obtaining the following functions:

  - $\sigma_{d_0}(\langle s_1, s_2\rangle) = \sigma_{d_2,i}(s_2)$
  - $\tau_{d_0}(\langle s_1, s_2\rangle) = \{s_1\} \times \tau_{d_2,i}(s_2, v)$
  - $o_{d_0}(\langle s_1, s_2\rangle) = o_{d_1}(s_1) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2) = \{s_1\} \times o_{d_2}(s_2)$

  The first two items use the fact that u_1 has no incident arcs, and the last, the fact that $o_{d_1}(s) = \emptyset$.

- Because d_1 enters into no transaction that would change its state, and because $o_{d_1}(s) = \emptyset$, the initial state of u_1 remains forever unchanged, i.e., s_1 in the above equations is a constant. Thus there is a 1-1 correspondence of states between u_2 and u_0, given by $s_2 \leftrightarrow \langle s_1, s_2\rangle$. Hence, we can map the state on u_0 back to what it would have been on u_2, and revert to the original σ_{d_2}, τ_{d_2}, and o_{d_2}, and have a 1-1 correspondence in execution.

In effect, no trace of u_1 remains. Of course, an implementation would delete u_1 directly, and not literally go through the steps that justify doing so.

As with ignoring an arc, one is not going to write boring activity descriptions on purpose; their utility arises in connection with projection and other transformations. In the next section we give an example that combines projection and deletion of both arcs and nodes.

4.2 Subgraph Extraction

The goal of this section is to "understand" a subgraph G_0 of a given graph G on its own terms, where by "on its own terms" we mean that we wish to leave G_0 itself untouched, and to view transactions along arcs leaving it as contributing to the non-determinacy of execution in the subgraph. More formally, our strategy is to define a projection on G which is the identity on nodes in G_0. The question then is, what happens to nodes outside G_0? We first consider the special case in which such a node has only one arc, the other end of which touches a node in G_0, and has $(n : p)$ as the signature for its activity description. In order to make the projection independent of the graph outside G_0, what we need is the *universal activity description for protocol p with name n*, denoted $d_{p,n}$. The definition is a bit technical, but the idea is simple: $d_{p,n}$ always responds to a sequence of transactions in a legal way, but is unpredictable up to the constraints imposed by p. The easy part is this:

- The signature for $d_{p,n}$ is $(n : p)$.
- The other constituents depend only upon p, and will be denoted $\mathcal{S}_p$, σ_p, τ_p, and o_p.
- $o_p(s) \overset{\text{def}}{=} \emptyset$ for all $s \in \mathcal{S}_p$.

The technicalities for the definitions of $\mathcal{S}_p, \sigma_p$, and τ_p are in Appendix A, where it is shown that there is a projection from d to $d_{p,n}$.

Generalizing the situation, suppose that a node u not in G_0 has activity description d with signature $(n_i : p_i)_i$. By "i touches G_0", we mean that the other end of the arc whose u-relative name is n_i touches a node in G_0. The projection for u takes d to an activity description that behaves on arc i like the universal activity description for p_i if i touches G_0, and which otherwise ignores arc i. Formally:

- $\Pi((n_i : p_i)_i) \overset{\text{def}}{=} (n_i : \left\{ \begin{array}{ll} p_i & \text{if } i \text{ touches } G_0 \\ \text{true} & \text{otherwise} \end{array} \right\})$
- $\mathcal{S}_{\Pi(d)} \overset{\text{def}}{=} \bigtimes_{i \text{ touches } G_0} \mathcal{S}_{d_i}$
- $\Pi(\sigma_{d,i})(\langle s_j \rangle_{j \text{ touches } G_0}, v) \overset{\text{def}}{=} \begin{cases} \sigma_{p_i}(s_i) & \text{if } i \text{ touches } G_0 \\ 0 & \text{(or any other arbitrary fixed value) otherwise} \end{cases}$
- $\Pi(\tau_{d,i})(\langle s_j \rangle_{j \text{ touches } G_0}, v) \overset{\text{def}}{=} \begin{cases} \bigtimes_{j \text{ touches } G_0} \left\{ \begin{array}{ll} \tau_{p_i}(s_i, v) & \text{if } j = i \\ s_j & \text{otherwise} \end{array} \right\} & \text{if } i \text{ touches } G_0 \\ \emptyset & \text{otherwise} \end{cases}$
- $\Pi(o_d)(\langle s_j \rangle_{j \text{ touches } G_0}) \overset{\text{def}}{=} \emptyset$

We omit the proof that there is a projection from d to the activity description with these components.

Now let us examine the result of this projection. We first observe that an arc between two nodes not in G_0 is ignored by both nodes, and hence may be deleted, by the previous section. After deleting all the arcs, any nodes that are not adjacent to a node in G_0 will be

4.2 Subgraph Extraction

The goal of this section is to "understand" a subgraph G_0 of a given graph G on its own terms, where by "on its own terms" we mean that we wish to leave G_0 itself untouched, and to view transactions along arcs leaving it as contributing to the non-determinacy of execution in the subgraph. More formally, our strategy is to define a projection on G which is the identity on nodes in G_0. The question then is, what happens to nodes outside G_0? We first consider the special case in which such a node has only one arc, the other end of which touches a node in G_0, and has $(n : p)$ as the signature for its activity description. In order to make the projection independent of the graph outside G_0, what we need is the *universal activity description for protocol p with name n*, denoted $d_{p,n}$. The definition is a bit technical, but the idea is simple: $d_{p,n}$ always responds to a sequence of transactions in a legal way, but is unpredictable up to the constraints imposed by p. The easy part is this:

- The signature for $d_{p,n}$ is $(n : p)$.
- The other constituents depend only upon p, and will be denoted $\mathcal{S}_p$, σ_p, τ_p, and o_p.
- $o_p(s) \overset{\text{def}}{=} \emptyset$ for all $s \in \mathcal{S}_p$.

The technicalities for the definitions of $\mathcal{S}_p, \sigma_p$, and τ_p are in Appendix A, where it is shown that there is a projection from d to $d_{p,n}$.

Generalizing the situation, suppose that a node u not in G_0 has activity description d with signature $(n_i : p_i)_i$. By "i touches G_0", we mean that the other end of the arc whose u-relative name is n_i touches a node in G_0. The projection for u takes d to an activity description that behaves on arc i like the universal activity description for p_i if i touches G_0, and which otherwise ignores arc i. Formally:

- $\Pi((n_i : p_i)_i) \overset{\text{def}}{=} (n_i : \left\{ \begin{array}{ll} p_i & \text{if } i \text{ touches } G_0 \\ \text{true} & \text{otherwise} \end{array} \right\})$
- $\mathcal{S}_{\Pi(d)} \overset{\text{def}}{=} \underset{i \text{ touches } G_0}{\times} \mathcal{S}_{d_i}$
- $\Pi(\sigma_{d,i})(\langle s_j \rangle_{j \text{ touches } G_0}, v) \overset{\text{def}}{=} \begin{cases} \sigma_{p_i}(s_i) & \text{if } i \text{ touches } G_0 \\ 0 & \text{(or any other arbitrary fixed value) otherwise} \end{cases}$
- $\Pi(\tau_{d,i})(\langle s_j \rangle_{j \text{ touches } G_0}, v) \overset{\text{def}}{=} \begin{cases} \times_{j \text{ touches } G_0} \left\{ \begin{array}{ll} \tau_{p_i}(s_i, v) & \text{if } j = i \\ s_j & \text{otherwise} \end{array} \right\} & \text{if } i \text{ touches } G_0 \\ \emptyset & \text{otherwise} \end{cases}$
- $\Pi(o_d)(\langle s_j \rangle_{j \text{ touches } G_0}) \overset{\text{def}}{=} \emptyset$

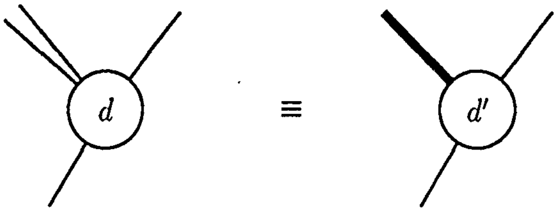
We omit the proof that there is a projection from d to the activity description with these components.

Now let us examine the result of this projection. We first observe that an arc between two nodes not in G_0 is ignored by both nodes, and hence may be deleted, by the previous section. After deleting all the arcs, any nodes that are not adjacent to a node in G_0 will be boring, and thus may be deleted, again by the previous section. Finally, any remaining node not in G_0 will have arcs to one or more nodes in G_0. If the number of arcs is greater than 1, it may be unpinched[2] until all nodes not in G_0 have a single arc that touches a node in G_0. This is the last step of the construction—we now have a graph that embeds G_0 but is completely independent of the graph structure of G not in G_0.

The purpose of pursuing this is example is to offer evidence of the power of the pinching, shrinking and projection operations. Starting with an informal notion of wanting to "understand" a subgraph of a graph, the exercise of formalizing this in terms of the operations of section 2.3 and 2.4 leads inevitably to the necessity of constructing an activity description that behaves as the most general participant in a protocol, which we called the universal activity description for a protocol. If we want merely to look at a subgraph of a graph, it would be overkill to invoke all this machinery. However, if we want to simulate or analyze a subgraph, or if we want to test and debug an activity protocol with a non-empty signature, an implemented version of $d_{p,n}$ is exactly what we need. So the seemingly technical device that we used to formalize extracting a subgraph from a graph turns out to have important connections to reality.

4.3 Collapsing Parallel Arcs

Section 2.3 showed how to condense any subgraph of a graph to a single node u_0 without loops. Consider a node u_1 outside the subgraph, which in the original graph had arcs to distinct nodes in the subgraph. In the transformed graph, the nodes u_0 and u_1 will be connected by several arcs. Arcs that are incident on the same pair of nodes are called *parallel*; the purpose of this section is to describe how parallel arcs can be collapsed to a single arc, and the activity descriptions on the nodes adjusted so that there is still a 1-1 correspondence in executions. The essence of the transformation is on an activity description:



The heavy arc on the right is obtained by collapsing the two arcs near each other on the left. Let the activity description on the left be d with signature $(n_i : p_i)_i$. For convenience, let n_0 be the name for the heavy arc. (Choose $n_0 \neq n_i, i > 2$.) The signature for the new activity description d' will be $(n_0 : p_0, n_3 : p_3, \ldots, n_k : p_k)$, where we have no vet defined p_0. Before doing so, we note that d' inherits the set of states from d, the definitions of $\sigma_{d,i}$ and $\tau_{d,i}$ for $i > 2$, and the definition of o_d.

The idea is that a value seen along the collapsed arc consists of the pair of values seen along the two uncollapsed arcs. There are two technical assumptions that we must make: the first is that $V \times V \subseteq V$, i.e., if $v_1, v_2 \in V$, then $\langle v_1, v_2 \rangle \in V$. The second has to do

[2] The pinching and shrinking transformations are equivalences, and may be applied in either direction, by "un-", we mean in the direction opposite to the one stated.

with a corresponding operation on protocols. Given $p_1, p_2 \in P$, we assume that there is a $p_1 \times p_2 \in P$ with the following action as a predicate:

- $(p_1 \times p_2)(\langle\langle f_i, v_i\rangle\rangle_i) \overset{\text{def}}{\Leftrightarrow}$ every v_i can be written in the form $\langle v_{i1}, v_{i2}\rangle$, and $p_j(\langle\langle f_i, v_{ij}\rangle\rangle_i)$ holds for $j = 1$ and 2.

We note that $(p_1 \times p_2)^T = p_1^T \times p_2^T$, i.e., they have the same action as predicates.

Naturally, we let $p_0 \overset{\text{def}}{=} p_1 \times p_2$, thus completing the signature for d', and to complete the definition of the entire activity description:

- $\sigma_{d',0}(s) \overset{\text{def}}{=} \langle \sigma_{d,1}(s), \sigma_{d,2}(s)\rangle$
- $\tau_{d',0}(s, \langle v_1, v_2\rangle) \overset{\text{def}}{=} \tau_{d,1}(s, v_1) \cup \tau_{d,2}(s, v_2)$
- $o_{d'}(s) \overset{\text{def}}{=} o_d(s)$

Result 7 Let u_1 and u_2 be nodes in a graph with activity descriptions d_1 and d_2, and suppose that a_1 and a_2 are parallel arcs adjoining them; for convenience, assume that a_i is the i^{th} arc on each node. Obtain a new graph by collapsing a_1 and a_2 to a single arc a_0, and by replacing the activity description on u_i by d'_i, as outlined above. Then there is a 1-1 correspondence between the execution of the two graphs.

Proof We can view this as a sequence of elementary transformations:

- Pinch u_1 and u_2, then shrink a_1 and a_2, now loops, obtaining the following functions for the new node u_0.
  - $\sigma_{0,a}(\langle s_1, s_2\rangle)$ for $a \neq a_1, a_2$ carries over from $\sigma_{i,a}(s)$.
  - Similarly for $\tau_{0,a}(\langle s_1, s_2\rangle, v)$.
  - $$\begin{aligned} o_0(\langle s_1, s_2\rangle) = & \quad o_1(s_1) \times \{s_2\} \cup \{s_1\} \times o_2(s_2) \\ & \cup\ \tau_{1,a_1}(s_1, \sigma_{2,a_1}(s_2)) \times \{s_2\} \cup \{s_1\} \times \tau_{2,a_1}(s_2, \sigma_{1,a_1}(s_1)) \\ & \cup\ \tau_{1,a_2}(s_1, \sigma_{2,a_2}(s_2)) \times \{s_2\} \cup \{s_1\} \times \tau_{2,a_2}(s_2, \sigma_{1,a_2}(s_1)) \\ = & \quad o_1(s_1) \times \{s_2\} \cup \{s_1\} \times o_2(s_2) \\ & \cup\ \tau_{d,0}(s_1, \sigma_{d',a_0}(\langle s_1, s_2\rangle)) \times \{s_2\} \\ & \cup\ \{s_1\} \times \tau_{d',a_0}(s_2, \sigma_{d',a_0}(\langle s_1, s_2\rangle)) \end{aligned}$$
- Unshrink a new arc a_0 and then unpinch to get nodes u'_1, u'_2 with:
  - $\sigma_{u_i,a}(s_1) = \begin{cases} \sigma_{i,a}(s_i) & \text{if } a \neq a_1, a_2 \\ \sigma_{d',a_0}(s_i) & \text{if } a = a_0 \end{cases}$
  - Similarly for $\tau_{u'_i,a}$
  - $o_{u'_i}(s_i) = o(s_i)$

The protocol $p_1 \times p_2$ has clearly been constructed to be valid for the communication on a_0.
□

Thus, parallel arcs can always be collapsed, assuming that $V \times V \subseteq V$ and that P is closed under the product operations discussed above.

We state without proof several relationships that collapsing arcs has with other operations. First, shrinking and collapsing commute:

Result 8 Let i_1, i'_1, i_2, i'_2 be distinct indices of the signature of activity description d, and form d' by collapsing i_1 and i_2, and i'_1 and i'_2, calling the result indices i_0 and i'_0. Then:

$$d - \{[i_1, i'_1], [i_2, i'_2]\} \cong d' - [i_0, i'_0]$$

□

Result 9 Let i_1, i_2, and i_3 be distinct indices of the signature of activity description d, form $d_{\langle 1,2\rangle,3}$ by collapsing i_1 and i_2, and the resulting dangling arc with i_3, and form $d_{1,\langle 2,3\rangle}$ by collapsing i_2 and i_3, and then collapsing i_1 with the resulting dangling arc. Then:

$$d_{\langle 1,2\rangle,3} \cong d_{1,\langle 2,3\rangle}$$

□

At the beginning of this section, we remarked that collapsing parallel arcs is useful in tidying up a graph which has had a subgraph condensed to a node. In an implementation this might be done automatically. In the next section, we shall see another use for this graph transformation.

4.4 Products of Isomorphic Graphs

A common pattern of activity is the assignment of essentially the same task to several persons, for example, all members of a committee are to receive a report and submit a review by a certain date. It is quite natural to describe what a committee member does from the point of view of one member, but from the point of view of the person who assigns the task to the committee and who collects the reviews, it is natural to look at one graph that summarizes the behavior of all the members of the committee.

The notion of projected execution (section 2.4) supplies precisely the right technique for obtaining from the single graph, the view that is specific to a particular committee member. What is needed is a way to obtain the single transaction graph that captures the behavior of a set of participants from a graph that specifies the behavior of a single participant, and does so in such a way that it is possible project the appropriate view for each particular participant.

As the title of this section suggests, we will formalize the notion of "essentially the same task" to mean that the activity descriptions are based on isomorphic graphs. We do not require that the activity descriptions on corresponding nodes be in any way related, but we will assume that arcs which are paired by the isomorphism have the same names—this is not a real restriction, since renaming is always possible. The details:

- Let G_1 and G_2 be isomorphic transaction graphs. This product, denoted $G_1 \times G_2$, is constructed as follows:
  - Pinch each pair of nodes that is paired by the isomorphism.
  - For each pair of dangling arcs paired by the isomorphism, transform the activity description on the node as described in section 4.3.
  - For each pair of non-dangling arcs paired by the isomorphism, collapse the arcs as described later on in the same section.
  - Let the signature of G_i be $(n_j : p_{ij})_j$; the signature of $G_1 \times G_2$ is evidently given by $(n_j : p_{1j} \times p_{2j})_j$.

We state without proof several desirable properties of this construction. The first says that the product is associative and commutative, so that we can write $G_1 \times G_2 \times G_3$ without ambiguity, and similarly $\times_{i \in I} G_i$, where I is an index set. Further, there is an identity element, so the latter makes sense even if $I = \emptyset$.

Result 10 Let G_1, G_2, G_3 all have isomorphic graph structure. Define 1_G to have the same graph structure, where the protocol on a vertex is the boring protocol with the appropriate signature. Then:

$$G_1 \times G_2 \cong G_2 \times G_1 \qquad G_1 \times (G_2 \times G_3) \cong (G_1 \times G_2) \times G_3 \qquad G_i \times 1_G \cong G_i$$

□

Product and projection work in the way one expects:

Result 11 Let G_1, G_2 be transaction graphs such that $G_1 \times G_2$ is defined. Let:

- $\Pi_i(n_j : p_{1j} \times p_{2j})_j \overset{\text{def}}{=} (n_j : p_{ij})_i$
- $\Pi_i(d_1 \times d_2) \overset{\text{def}}{=} d_i$
- $\Pi_i(\langle s_1, s_2 \rangle) \overset{\text{def}}{=} s_i$
- $\Pi_i(\langle v_1, v_2 \rangle) \overset{\text{def}}{=} v_i$

Then $\Pi_i(G_1 \times G_2) \cong G_i$, for $i = 1$ and 2.

□

Finally, there is a natural connection between graphical collapse and products of protocols and graphs.

Result 12 Let G_1, G_2 be as above, and for any G, let d_G be the transaction protocol obtained by collapsing G to a single vertex. Then:

$$d_{G_1 \times G_2} \cong d_{G_1} \times d_{G_2}$$

□

4.5 Exclusive Access

Because of the strong desire to achieve a distributed system, transaction graphs have been defined to be quite asynchronous—each node talks to only one neighbor at a time, and there is no central overseer of the communication. And even this communication does not require the active participation of the neighbor—in fact the main restriction on this communication is that the neighbor not be *too* active, i.e., trying to communicate at the same time.

While the implementation basis for an activity coordination system is necessarily distributed and thus provides only loose coupling among the constituents, it is also the case that a rather fundamental form of activity "coordination" requires tighter coupling, namely, exclusive access—ensuring that among a subset of competitors wanting access to a resource, permission will be granted to only one at a time. In a distributed system, we can expect that non-adjacent nodes may concurrently arrive at a state in which they need exclusive access to a resource, but of course, a node cannot unilaterally decide that it has such access. There may be several ways to pose the problem in a formal way; the one we choose is based on a classical two-phase notion: a node announces that it *would like* exclusive access; some communication ensues, which results in the node's being told that all other nodes agree that it can have exclusive access, or not. If exclusive access is denied, the node may re-try, but it may also decide, on the basis of the new information it received that it is no longer interested in exclusive access. This section will provide one formalization of what exclusive access might mean, and will provide one implementation which gives rise to the desired behavior. There are no doubt both other definitions of exclusive access and other implementations that would meet the definition we have chosen, so the goal here is not to provide the ultimate technique for exclusive access, but rather to give an example of the use of transaction graphs, admittedly at a rather low level, for a very real and practical problem.

For simplicity, we will assume that all the nodes of the graph are potential competitors. In practice, this would more likely be a subset of nodes; the subset might even be dynamically determined. The definition will use the idea that a node goes through phases. Each node starts at phase 0, and the phases are numbered sequentially. (The phase does not have to be stored in the node's state; as we will see, it is assigned only as part of our observation of the node's behavior.) The specification for the synchronization is as follows:

- A node in an even phase may enter an odd phase in one of two ways:
  - It may announce that it would like exclusive access.
  - It may see that some other node has entered an odd phase.

  Note that an odd phase does not start until some node wants exclusive access, i.e., no polling is necessary.

- For all odd k, no node enters phase $k+1$ until all nodes have entered phase k.

- For all odd k, exactly one node receives exclusive access.

The solution we propose is based on the idea of a priority for the different nodes, where the priority is used to resolve competition among nodes during a given phase. We shall initially assume that each node has a distinct priority, but after discussing the solution, will discuss techniques that allow the assumptions to be weakened. A priority is a positive integer. Values exposed along arcs will be non-negative integers, and will alternate between zero and positive values. The driving idea is quite simple:

- To indicate that it would like exclusive access, a node begins the propagation of its priority throughout the graph.
- Nodes with higher priority that are not interested in exclusive access during this phase, and all nodes with lower priority, assist in the propagation.
- Eventually, a node wanting exclusive access will detect that permission has been granted, or will find itself overrun by a higher priority propagation.

There are probably several versions of a propagation algorithm; the one presented here is based on a simple algorithm for marking a connected component of a graph. A node is considered "marked" with priority l if it is exposing l on an arc. A node wanting exclusive access attempts to put its priority on each of its arcs, thereby marking itself with its own priority. It waits to see whether the propagation is successful and eventually finds out whether it has been granted access. If so, it can perform the desired action, and it then releases its exclusive access.

To make all of this completely precise, we describe the set of states for an activity description d and the definitions of $\sigma_{d,i}$ and $\tau_{d,i}$.

- For a node with k arcs, its state will be a k-tuple of triples, each consisting of two integers and a flag, denoted $v_i/v'_i/f_i$.
  - v_i is the value exposed along the i^{th} arc.
  - v'_i is the last value seen in a transaction with the i^{th} neighbor.
  - If $v_i \neq 0$, then $f_i = 1$ if during this phase, the neighbor's exposed value was first to become equal to $\max(v_i, v'_i)$; otherwise $f_i = 0$.
  - If $v_i = 0$ and $v'_i \neq 0$, then $f_i = 1$ perforce.
  - If $v_i = 0$ and $v'_i = 0$, then $f_i = 1$ means that the node is almost ready to re-enter the initial state (see the next item).
- Initially, a state is $\langle 0/0/0 \rangle_j$.

All of the activity descriptions will be the same except for variability in the number of arcs, and the fact that each d has an innate priority l_d.

In presenting the definition of $\tau_{d,i}$ for a node in state $\langle v_j/v'_j/f_j \rangle_j$, we shall use the convention that $l \overset{\text{def}}{=} \max_j(\max(v_j, v'_j))$. Initially, $l = 0$ at every node, and in this graph state the following transaction is the only way for l to become positive at any node.

- $\tau_{d,i}(\langle 0/0/0 \rangle, 0) \overset{\text{def}}{=} \{\langle \ldots, l_d/0/0, \ldots \rangle\}$, leaving triples other than the i^{th} unmodified (this convention applies in all definitions of transactions).

This transaction evidently exposes its innate priority on an arc.

Once a priority is exposed on an arc, a neighboring node can increase its value of l with the following transaction:

- $\tau_{d,i}(\langle v_j/v'_j/f_j\rangle_j, v''_i) \overset{\mathrm{def}}{=} \{\langle \ldots, v_i/v''_i/1, \ldots\rangle\}$, where
  - $l < v''_i$ (the node did not know about this high a priority).

These are the only two transactions that can increase the value of l at a node.

The next transaction propagates a priority to a neighbor that may not have seen it yet.

- $\tau_{d,i}(\langle v_j/v'_j/f_j\rangle_j, v''_i) \overset{\mathrm{def}}{=} \{\langle \ldots, l/v''_i/0, \ldots\rangle\}$
  - $v_i < l$ (the node has seen a higher priority but not exposed it on this arc).
  - $v''_i < l$ (the neighbor may not know about priority l).

Suppose we apply the transactions stated thus far until none of them applies any more. Then in state $\langle v_j/v'_j/f_j\rangle_j$, we will either have $v_i = l, v'_i < l$ and $f_i = 0$, or $v_i < l, v'_i = l$, and $f_i = 1$. The latter case does not occur in the node whose innate priority is l, and occurs in all other nodes exactly once (on the arc along which it first saw l). Imagine orienting all arcs toward the end where $f_i = 1$ (this can't be the case at both ends). The oriented arcs form a spanning tree of the transaction graph, where all paths in the tree lead toward the node whose innate priority is l.

It is easy for the root of the tree to tell that it is the root—it is the only node where $v_i = l$ and $f_i = 0$ for all i. The hard part is this: how does the root know that the above transactions have been done until none of them is possible? Let us first make a simplifying assumption that the transaction graph is a tree. Then it is also easy for every leaf to tell that it is a leaf. It can announce that propagation can go no further by exposing l on the arc where $f_i = 1$. When nodes further up the tree see l on arcs pointing to them, they can in turn expose l on their arcs where $f_i = 1$, until eventually, the root node sees itself surrounded by l, at which point is knows it has excluded all other nodes.

Let us drop the assumption that the transaction graph has no cycles. If the above strategy is pursued, a block arises because a leaf cannot detect that it is a leaf—it may still see values less than l, specifically, on cycle arcs. But cycle arcs are fortunately easy to detect—when a node sees its l on an arc where $v'_i < l$, it knows the arc is involved in a cycle, and to break the cycle, exposes l on it, using the following transaction

- $\tau(\langle v_j/v'_j/f_j\rangle_j, l) \overset{\mathrm{def}}{=} \{\langle \ldots, l/l/1, \ldots\rangle\}$, where
  - $v'_i < l$ (the node knows about l, but l arrived at this neighbor by some other route).

Again, suppose we apply the transactions stated thus far until none of them applies. Then in state $\langle v_j/v'_j/f_j\rangle_j$, we will have $v_i = l, v'_i < l$ and $f_i = 0$ (the arc is an incoming tree arc), or $v_i = l, v'_i = l$ and $f_i = 0$ (the arc is a cycle arc, which we also call incoming), or $v_i < l, v'_i = l$ and $f_i = 1$ (the arc is an outgoing tree arc), or $v_i = l, v'_i = l$, and $f_i = 1$ (the arc is a cycle arc, which we call outgoing). In this state, a leaf of the spanning tree will have $f_i = 1$ on its outgoing tree arc, and all other arcs (if any) will be incoming cycle arcs on which the leaf will have seen l. Thus the following rule will cause it to expose l on its outgoing tree arc.

- $\tau(\langle v_j/v'_j/f_j\rangle_j, l) \overset{\text{def}}{=} \{\langle \ldots, l/l/1, \ldots\rangle\}$, where
  - $v_i < l, v'_i = l$ and $f_i = 1$ (l not previously exposed on the tree arc).
  - For $j \neq i, v'_j = l$ (all other arcs are cycle arcs or incoming tree arcs, and have been marked).

This rule causes l to propagate up the tree, and eventually the state of every node becomes $\langle l/l/f_j\rangle_j$. It is in this state that the root node, where $l = l_d$, has exclusive access. After it has done its business, it begins setting exposed values on incoming arcs (there must be at least one) to 0, using the following transaction, which also is partly responsible for propagating 0 against the imagined direction of the arcs:

- $\tau_{d,i}(\langle v_j/v'_j/f_j\rangle_j, l) \overset{\text{def}}{=} \{\langle \ldots, 0/l/1, \ldots\rangle\}$, where
  - Either $l = l_d$ or $v'_j = 0$ for some j.
  - $v_i = l, v'_i = l$ and $f_i = 0$.
  - For all $j \neq i, v_j = l, v'_j = l$ and $f_j = 0$ (this transaction hasn't yet happened), $v_j = 0, v'_j = l$ and $f_j = 1$ (it has happened), or $v_j = l$ and $f_j = 1$ (this is an outgoing arc).

To trigger this transaction in nodes other than the root, it is necessary and sufficient to see 0 along a forward arc.

- $\tau_{d,i}(\langle v_j/v'_j/f_j\rangle_j, 0) \overset{\text{def}}{=} \{\langle \ldots, l/0/1\rangle\}$, where
  - $v_i = l, v'_i = l$, and $f_i = 1$

Stabilizing with these transactions results in the imagined direction of arcs going from an exposed value of 0 to an exposed value of l.

The final transaction applies when a node has seen 0 along all incoming arcs; it then exposes 0 along what were previously considered outgoing arcs, destroying the last record of the orientation:

- $\tau_{d,i}(\langle v_j/v'_j/f_j\rangle, 0) \overset{\text{def}}{=} \{\langle \ldots, 0/0/1, \ldots\rangle\}$, where
  - $v_i = l, v'_i = 0$, and $f_i = 1$
  - For $j \neq i, v'_j = 0$

The final step is not a transaction, but an out-of-graph change that resets the state to the initial one.

- $o_d(\langle 0/0/1\rangle_j) \overset{\text{def}}{=} \{\langle 0/0/0\rangle\}$

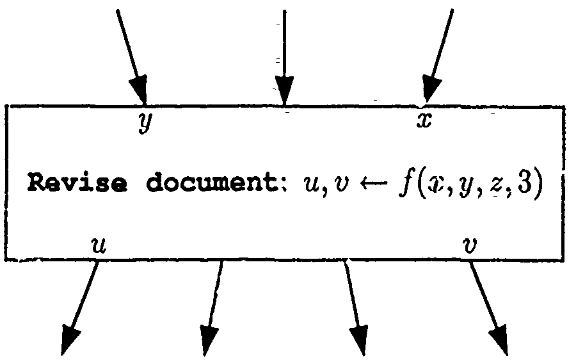
Note that this step may occur and subsequent phases begin in nodes other than the root before the root has entirely reset itself, but this causes no harm, because once an arc has the form 0/0/1, no transaction occurs along it until the node is entirely reset.

We close with the promised remarks on priority. The first point is that the priority need not be innate in the activity description but instead can be part of the state; further, the priority relationships do not have to be the same from phase to phase. For example, if n is the number of nodes in the graph, as each node resets itself at the end of a phase, it could change l_d to $l_d+1 \bmod n$, so that there is a rotating system of priorities. Other possibilities include data dependent priorities with l_d used only as a tie-breaker. Still another possibility is to use probabilistically computed priorities, accompanied merely by a unique id, and in those cases where two nodes happened to assign themselves the same priorities, tolerate a certain probability that no transaction will be performed during a phase. The point here is not so much what scheme is used, but that the scheduling and even the degree of indeterminacy is in the hands of the protocols, not of the system. There is no assumption of fair scheduling, indeed, we can assume a malicious scheduler.

4.6 Procedure-Based Activity Descriptions

Suppose we have procedure f that takes some number of arguments and yields some number of results. We explicitly allow f to perform out-of-graph communication, for example, accessing and updating an external database. In this section, we discuss the *activity description based on* f. The idea is that f lives inside a node whose incoming arcs both supply some of the arguments and determine, by being *enabled*, when f is to be invoked. Dually, the outgoing arcs say when the application of f is complete, i.e., are enabled, and may be used to propagate results to neighbors.

In a parameterized transaction graph, a reasonable convention would be to reserve rectangles for procedure-based activity descriptions, thus devoting a simple shape to what will probably be the most common form of activity description. An example:



The "**Revise document**" is the label that will appear on the corresponding node in a participant's view of an activated graph. Note that some of the parameters supplied to f may be local, i.e., arc labels; others may be non-local, and refer to parameters of the containing parameterized transaction graph; the other possibility is a constant. An unlabeled incoming arc is one whose only contribution is the fact that it is enabled; more information than that about the arriving value doesn't matter. Similarly, the value sent down an unlabeled output arc is one which has no information beyond the fact that it is sent.

The reader may have noticed the directed arcs, in contrast to our usual assumption that the transaction graph has undirected arcs. This is not a change to the theory, but reflects the fact that the above node is at a more concrete level than what we have been discussing—it is what might actually be seen in a graphical display of a parameterized transaction graph. Moreover, the directions on the arcs do carry meaning, as we now explain. The arcs on a procedure-based activity description have one of two basic forms of protocols, "send" and "receive", and the directions of the arcs indicate which, in the natural way. Let us suppose that our underlying language has types. The protocols are given by:

- $\texttt{send}(t)$ and $\texttt{receive}(t)$, where $(\texttt{send}(t))^T = \texttt{receive}(t)$.

The protocols specify exactly what is meant by "enabled" (used informally above). One way to do this is as follows:

- $\texttt{send}(t)$ is true of $\langle\langle f_i, v_i\rangle\rangle_i \overset{\text{def}}{\Leftrightarrow}$
  - If $f_i = 0$, then v_i (the value being "sent") is either $\langle \mathit{false}, \mathit{nil}\rangle$, indicating that the arc is not enabled, or is $\langle \mathit{true}, w\rangle$, where w is a value of type t, indicating that the arc is enabled.
  - If $f_i = 1$, then v_i (the reply) is either *false* (ready for a value to be sent), or *true*.
  - If $\langle\langle f_i, v_i\rangle\rangle_i$ is reduced, then f_i alternates between 0 and 1 (once a value is exposed, it is held until a change is seen), and the sequence of $\langle v_i\rangle_i$ is of the form:

$$\ldots\ \langle \mathit{true}, w_1\rangle\ \mathit{true}\ \langle \mathit{false}, \mathit{nil}\rangle\ \mathit{false}\ \langle \mathit{true}, w_2\rangle\ \mathit{true}\ \langle \mathit{false}, \mathit{nil}\rangle\ \ldots$$

It is not necessary to define how $\texttt{receive}(t)$ acts as a predicate, since that is determined by $\texttt{send}(t)$.

Earlier, we gave an informal description of how a procedure-based activity description works. We now state more precisely how one works, not only guaranteeing the behavior specified by the above protocols, but also showing the details of how inputs and outputs are coordinated. As with mutual exclusion, it is useful to think of one of these nodes as going through phases, but unlike mutual exclusion, one wants to decouple behavior of nodes, up to the necessity of coordinating various inputs and outputs.

- During a "transmit" phase, values on incoming arcs change from *false* to *true*, acknowledging (and grabbing) an argument value; values on outgoing arcs change from $\langle \mathit{false}, \mathit{nil}\rangle$ to $\langle \mathit{true}, \ldots\rangle$, posting a result value.

- During a "compute" phase, values on incoming arcs change from *true* to *false*; values on outgoing arcs, from $\langle \mathit{true}, \ldots\rangle$ to $\langle \mathit{false}, \mathit{nil}\rangle$. Both these may be considered "resets".

Let $(n_i : \texttt{receive}(t_i))_i \cdot (n'_i : \texttt{send}(t'_i))_{i'}$ be the protocol for a procedure-based activity description. Then the state has:

- A value on each arc to buffer the corresponding argument or result, denoted x_i and x'_i, respectively.

- A boolean on each arc to control communication there, denoted b_i and b'_i for the two sets of arcs.
- A single boolean to record compute/transmit state, denoted c, for "compute" when *true*.
- As a notation, the state $s \overset{\text{def}}{=} \langle c \rangle \frown \langle\langle b_i, x_i \rangle\rangle_i \frown \langle\langle b'_i, x'_i \rangle\rangle_{i'}$

Exposed values are controlled by:

- $\sigma_i(s) \overset{\text{def}}{=} b_i$
- $\sigma'_{i'}(s) \overset{\text{def}}{=} \begin{cases} \langle true, x'_i \rangle & \text{if } b'_{i'} \\ \langle false, nil \rangle & \text{otherwise} \end{cases}$

Values exposed on incoming arcs change to *true* only during the transmit phase, and only when an available value is seen:

- $\tau_i(s, \langle true, w_i \rangle) \overset{\text{def}}{=} \{\langle \ldots \langle true, w_i \rangle \ldots \rangle\}$, when:[3]
  - c is *false.*
  - b_i is *false.*

Dually, this is also when values exposed on outgoing arcs become $\langle true, x'_i \rangle$:

- $\tau'_{i'}(s, false) \overset{\text{def}}{=} \{\langle \ldots \langle true, x'_{i'} \rangle \ldots \rangle\}$, when:
  - c is *false.*
  - $b'_{i'}$ is *false.*

The end of a transmit phase is made official by the following:

- $o(s) \overset{\text{def}}{=} \{\langle true, \ldots \rangle\}$ when:
  - c is *false.*
  - For all i, b_i is *true*, and for all $i', b'_{i'}$ is *true.*

The reader may supply the transactions that reset the arcs during a compute phase. The end of the compute phase is marked by the change of state due to the application of the procedure, here called f.

- $o(s) \overset{\text{def}}{=} \{\langle false \rangle \frown \langle\langle b_i, x_i \rangle\rangle_i \frown \langle\langle b_i, x''_i \rangle\rangle_{i'}\}$ when:
  - c is *true.*
  - For all i, b_i is *false* and for all i', b'_i, is *false.*
  - $\langle x''_{i'} \rangle_{i'}$ denotes the results of applying f to $\langle x_i \rangle_i$.

[3] We follow a convention throughout that only change is indicated, in this case, to the values b_i, x_i.

Observe that this does not change values on outgoing arcs, in spite of the fact that it changes x'_i, because b'_i is *false*.

We can summarize the results of the above discussion by giving our first example of a parameterized activity description.

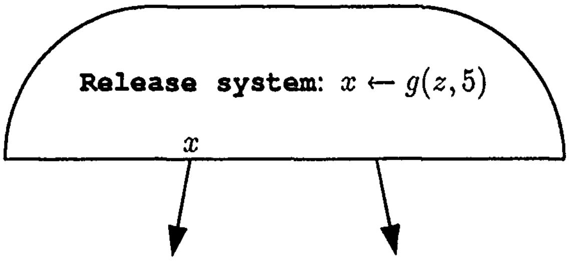
- `procedure-based-activity-description`
  string list(name × type) procedure list(name × type) $\longrightarrow$ *activity-description*
  - The string supplies the label for the participant's view.
  - Let $\langle\langle n_i, t_i\rangle\rangle_i$ and $\langle\langle n'_i, t'_i\rangle\rangle_i$ be the two *list(name × type)* arguments. Then the *procedure* argument takes arguments of types $\langle t'_{i'}\rangle_{i'}$.
  - The signature for the result is $(n_i : \texttt{receive}(t_i))_i \,\hat{}\, (n'_i : \texttt{send}(t'_i))_{i'}$.
  - The state and the functions σ, τ, and o are described above.

The reader may worry that in `procedure-based-activity-description`, the procedure argument f must exactly match the signature, while in the discussion of the graphical representation, there is a looser connection between `f` and the incident arcs. This discrepancy is handled by a translation step which produces f from `f`.

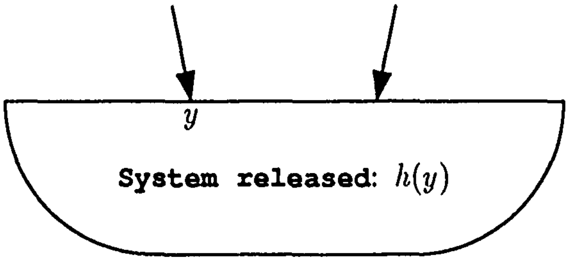
Let us consider a procedure-based activity description with no incoming arcs, initalized with $c = true$ and $b'_{i'} = false$. Then the only change of state will come from having applied f and thereby computed $\langle x'_{i'}\rangle_{i'}$, which are then propagated along outgoing arcs. Once this has happened and the arcs are all reset, the compute state is re-entered, and f is re-invoked. In other words, such an activity description will invoke f as often as it can, up to the rate at which its results are absorbed and it can supply them. This is ideal for generating repeated events, but not so good when we want an "initialization" node, whose procedure is invoked just once at the beginning of an activation. One could construct a procedure which delayed forever the second time it was invoked, but this seems artificial, and initialization is common. In the interests of conciseness of graphs, it might be reasonable to introduce a second kind of parameterized activity description, a slight variant on the previous.

- `procedure-based-activity-description-initialize`
  string procedure list(name × type) $\longrightarrow$ *activity-description*
  - The arguments are the same as before, with the omission of those for incoming arcs.
  - The state, σ, τ, and o are also the same as before, except:
    - $*$ c is omitted from the state, and in the values for τ given previously, the condition on c is omitted.
    - $*$ The rule for o that sets c to *true* is dropped.

In the spirit of using a distinct shape for each primitive parameterized activity description, the above might seen in parameterized transaction graphs as:



Dually, nodes which are expected to see exactly one phase of input might be drawn thus:



Consider a graph that has initialization and termination nodes, and where all nodes with both incoming and outgoing arcs have procedure-based activity descriptions. Consider further the case in which the graph is acyclic and in which it is possible to assign a number to each node which corresponds to how long the procedure takes, once it arguments are ready. Then we have a classic pert chart, and can use well-known techniques to do, say critical path analysis. Such an analysis is not deep of course, but the point is that this is a simple example of tying a transaction graph to an analysis tool. Transaction graphs are integrated into the everyday activities of its users, and thus provide a way of connecting the activities of its users with analysis, for example, monitoring whether the assumptions of the analysis become inconsistent with reality.

Suppose that a transaction graph consists entirely of procedure-based activity descriptions, and allow the graph to have cycles. Then the execution of a graph corresponds to a "firing sequence" in a marked graph, a subclass of Petri nets, studied extensively in [HC69]. There are a variety of analysis techniques, including liveness (deadlock-free-ness), maximum cyclic rates, and peak resource usage. As with the connection to pert charts, transaction graphs provide a connection between useful analysis techniques and the actual progress of a real activity.

In both the above examples, it should be remembered that it is not necessary that the detailed transaction graph used to coordinate everyday activities be precisely of the form required by the analysis technique. It is more likely that there is a projection that takes the detailed transaction graph to a graph that lies within the class, so that the analysis applies to a particular view of the activity, as one would expect.

A The Universal Activity Description for a Protocol

We show that for any protocol p, there exist $\mathcal{S}_p$, σ_p, τ_p, and o_p, which together with the signature $(n : p)$ comprise an activity description $d_{p,n}$ with the following property:

- For *any* activity description d with signature $(n : p)$, there is a projection from d to $d_{p,n}$.

Because of the generality of this result, in particular, the fact that p can be an arbitrary predicate, the proof we give is non-constructive—it does not say how to implement $d_{p,n}$. In practice, this is usually not difficult, but even if it is, the result says to keep trying, because the solution exists.

The almost right idea for an element of $\mathcal{S}_p$ is a sequence of pairs $\langle f_1, v_1\rangle, \ldots, \langle f_k, v_k\rangle$ for which p holds and which ends with $f_k = 0$. In this "state":

- $\sigma_p(\langle\langle f_i, v_i\rangle\rangle_{l=1}^{k})$ would be defined to be v_k.
- $\tau_p(\langle\langle f_i, v_i\rangle\rangle_{l=1}^{k})$ would be defined to be the set of all sequences $\langle\langle f_i, v_i\rangle\rangle_{i=1}^{k}$ where:
  - p holds for $\langle f_1', v_1'\rangle, \ldots, \langle f_l', v_l'\rangle$ and $f_l' = 0$ (otherwise this would not be a legitimate state).
  - The new sequence is an extension of the original, i.e., $l > k$ and $f_i' = f_i$ and $v_i' = v_i$ for $i = 1, \ldots, k$. Thus, we may drop the primes from f_i' and v_i'.
  - The new sequence does not skip any transactions at the node in question, i.e., $f_i = 1$ for $i = k+1, \ldots, l-1$. (If $l = k+1$, this is a vacuous condition.)
  - The new sequence is compatible with the value seen by τ, i.e., : $v = v_{i_0}$ where $i_o = \max\{i < l \mid f_i = 1\}$. (If $f_i = 0$ for all i, this is vacuous.)

The motivation behind this definition was stated earlier: τ_p responds to a sequence of transactions in a legal way, and it will respond to any legal sequence, i.e., is unpredictable up to the constraints imposed by p. The only reason that it is not quite right is that a state "remembers" too much: a sequence of states in an execution that we wish to project may repeat, but states in the above definition of τ always grow longer and thus never repeat. Hence we cannot obtain a projection.

This flaw can be remedied by using as states, i.e., elements of $\mathcal{S}_p$, not sequences of the above form, but rather, quotient sets of sequences of the above form, under the following equivalence relation:

- For $j = 1$ and 2, let $t_i \overset{\text{def}}{=} \langle\langle f_{ij}, v_{ij}\rangle\rangle_{l=1}^{k}$. We will assume that p holds for t_j and that $f_{k_j j} = 0$. We define $t_1 \sim t_2 \overset{\text{def}}{\Leftrightarrow}$
  - $v_{k_1,1} = v_{k_2,2}$ (both "states" expose the same value).
  - For any t_0, $p(t_1 \frown t_0) \overset{\text{def}}{\Leftrightarrow} p(t_2 \frown t_0)$ (where $\frown$ means concatenation).

Intuitively, p can't tell the difference between t_1 and t_2.