TASK: UR40
CDRL: 01210

(2)

UR40 — Repository Integration AdaKNET Software User's Manual

Informal Technical Data

UNISYS

STARS-RC-01210/002/00

24 October 1990

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
3 October 1990

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
Reusability Library Framework (RLF)
Ada Knowledge NETwork (AdaKNET) User's Manual

5. FUNDING NUMBERS
STARS Contract
F19628-88-D-0031

6. AUTHOR(S)
James J. Solderitsch
Ray McDowell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

8. PERFORMING ORGANIZATION REPORT NUMBER
GR-7670-1170(NP)

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Department of the Air Force
Headquarters, Electronic Systems Division (AFSC)
Hanscom AFB, MA 01731-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
01210 Volume II

11. SUPPLEMENTARY NOTES
There are two other related RLF reports:
(RLF) AdaTAU User's Manual and (RLF) Librarian Software User's Manual

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

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

AdaKNET (Ada Knowledge NETwork) is the semantic network subsystem of the RLF. AdaKNET enables the creation and modification of structured inheritance networks to represent detailed patterns of information. The manual outlines the current structure of the AdaKNET system and indicates how to make effective use of the available programmatic interfaces. A sample session description is included along with an appendix describing the basic information structuring primitives within AdaKNET. The manual also describes the specification language used to declare semantic networks for processing by AdaKNET.

Raymondo:
STARS (Software Technology for Adaptable Reliable Systems)

14. SUBJECT TERMS
Ada semantic network
Semantic Network Definition Language (SNDL), (KR)

15. NUMBER OF PAGES
45

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
SAR

TASK: UR40
CDRL: 01210

INFORMAL TECHNICAL REPORT

For The

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Repository Integration
AdaKNET Software
User's Manual

STARS-RC-01210/002/00
Publication No. GR-7670-1170(NP)
3 October 1990

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0002

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Unisys Defense Systems
Tactical Systems Division
12010 Sunrise Valley Drive
Reston, VA 22091

TASK: UR40
CDRL: 01210

INFORMAL TECHNICAL REPORT
For The
SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Repository Integration
AdaKNET Software
User's Manual

STARS-RC-01210/002/00
Publication No. GR-7670-1170(NP)
3 October 1990

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0002

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Unisys Defense Systems
Tactical Systems Division
12010 Sunrise Valley Drive
Reston, VA 22091

PREFACE

This document was prepared by Unisys Defense Systems, Valley Forge Operations, in support of the Unisys STARS Prime contract under the Repository Integration task (UR40). This CDRL, 01210, Volume II, is type A005 (Informal Technical Data) and is entitled "AdaKNET Software User's Manual".

This document has been reviewed and approved by the following Unisys personnel:

UR40 Task Manager: Richard E. Creps

Reviewed by: [illegible]
Richard E. Creps, System Architect (Acting)

Approved by: [illegible]
Hans W. Polzer, Program Manager

Table of Contents

Table of Figures

1. Scope

This document assumes that the user has a basic understanding of the Ada language and wishes to learn how to incorporate knowledge-based capabilities into a larger system. This document is not tutorial in nature with regard to the Ada language, nor does it cover basic material from the field of Artificial Intelligence (AI) whose study led to the development of the fundamental ideas that are implemented in the system described in this manual. The interested reader is referred to one of the many texts on Ada or AI; in particular, the Ada Language Reference Manual [LRM83] and *The Handbook of Artificial Intelligence*, Volume 1[Barr81].

1.1. Identification

This Software User's Manual provides a description of the content and basic operating procedures of AdaKNET, a subsystem level component of the Reusability Library Framework (RLF). Other major components of the RLF include AdaTAU and the Librarian application which are covered in separate user's manuals.

1.2. Purpose

The purpose of AdaKNET is to provide a system for representing structured domain knowledge. AdaKNET implements a knowledge representation formalism in the structured inheritance family. As such, it provides a framework for controlled evolution of a large body of knowledge, where the built-in constraints provided by the formalism help prevent the expression of meaningless or inconsistent models. In this way, there is a rough correspondence of the benefits provided by AdaKNET to domain modeling that strong typing provides to software development.

AdaKNET is implemented as layered abstract data types (ADTs), although users of AdaKNET need only "with" a single package (package AdaKNETs). That is, AdaKNET provides a programmatic interface to a semantic network model; operations are provided for creating, saving, restoring, manipulating, and examining the structure of AdaKNET instances. This manual describes the semantic network model implemented by AdaKNET, the overall architecture of AdaKNET (layered abstractions), and provides a detailed look at the top level AdaKNETs package. Also covered is a description of the Semantic Network Description Language (SNDL). Finally, a discussion on the pragmatics of installing and using AdaKNET and SNDL is provided.

1.3. Introduction

The remainder of this document is organized as follows. Section 2 lists a few RLF documents that have particular relevance to this user manual; other references are included in a bibliography. Section 3 provides an overview of the AdaKNET system. Included in this discussion is a summary of the model formalism supported by the AdaKNET implementation; a brief discussion of the overall architecture of the AdaKNET implementation is also provided, although users of AdaKNET need only be concerned with a single package, AdaKNETs. Section 4 provides a more detailed look at the AdaKNETs package; a brief synopsis of the operations provided by AdaKNET is provided. More details on each operation can be found in the actual package specification. Section 5 discusses SNDL, focusing on the language rationale, use, and major concepts. Section

6 discusses the pragmatics of installation and use of AdaKNET and SNDL to build knowledge-based applications in Ada; also included is a transcript of a session using the interactive AdaKNET browser-editor application. Finally, appendix A provides a detailed description of the AdaKNET conceptual model; users of AdaKNET should familiarize themselves with this definition. Appendix B provides a detailed definition of SNDL syntax and semantics (a syntax summary is also included).

2. Referenced Documents

In addition to the Ada LRM, and the AI Handbook referenced earlier, the following RLF documents are useful as references in conjunction with this document. Documents marked with an asterisk (*) were delivered to the Naval Research Laboratory as part of the original STARS Foundation contract (number N00014-88-C-2052) that supported the initial development of the RLF.

(*) Reusability Library Framework AdaKNET/AdaTAU Design Report.

(*) Gadfly User's Manual.

AdaTAU User's Manual.

Librarian User's Manual.

The RLF Librarian: A Reusability Librarian Based on Cooperating Knowledge-Based Systems [McDowell89].

The Growing of an Organon: A Hybrid Knowledge-Based Technology and Methodology for Software Reuse [Simos88].

Construction of Knowledge-Based Components and Applications in Ada [Wallnau88].

Constructing Domain-Specific Ada Reuse Libraries [Solderitsch89].

3. AdaKNET System Overview

AdaKNET is based partly on KL-ONE [Brachman85], and partly on K-NET, a Unisys-proprietary structured inheritance system. Although we provide a brief summary of the semantics of the underlying *conceptual model* implemented by AdaKNET, use of the AdaKNET ADT will require a more thorough understanding of the sometimes complex semantics of structured inheritance networks; appendix A of this manual provides an in-depth description of the AdaKNET conceptual model.

The AdaKNET implementation described here supports strict specialization (subsumption) semantics, range/value constraints on object attributes (Roles), single and multiple inheritance of attributes, and the subdivision of attributes (Subroles); the implementation also distinguishes generic classes of objects (Concepts) from instances of these classes (Individuals). These features provide sufficient modeling generality to describe a broad category of domain models for building knowledge-based applications (e.g. Gadfly, Librarian).

Instances of AdaKNET can be thought of as complex graphs (hence the term "semantic network"). AdaKNET views such graphs from an abstract data type (ADT) perspective. More specifically, AdaKNET implements an ADT for a class of data structures known as *heterogeneous, polylithic* structures [Booch87]; AdaKNET instances are composed of collections of component ADTs (heterogeneous), and instances of these component ADTs can be referenced from various access paths (polylithic).

Although the details of the underlying model may differ among various semantic network formalisms, they all share some common properties. A semantic network is a form of knowledge representation; it provides a means of denoting objects and describing relations that hold among them. The form of the representation can be thought of as a directed graph: a collection of vertices and edges. Each vertex in the semantic network denotes an object or a class of objects, and each edge describes a relation between objects and/or object classes. Typically, such networks support at least two forms of relationships between semantic objects: the specialization relation ("IS-A") and the aggregation relation ("IS-PART-OF"). A discussion of the need and use of these relations in domain modeling is extensive in database as well as artificial intelligence literature [Smith77].

The specialization relation indicates that one object class is a subset of another. For example, the class of objects consisting of all humans is a subset of the class composed of mammals. In semantic networks, all objects participate in a hierarchy (actually a semi-lattice, or DAG) of specialization relations between objects; this hierarchy is sometimes referred to as a *taxonomy*.

The aggregation relation indicates that one object can be considered a component or part-of another object. For example, a car consists of wheels, doors, and engine, etc. That is, a car is an aggregation of wheels, doors, and engine. The set of aggregation relations within a semantic network constitutes a subnetwork of the semantic network; not all objects need participate in this subnetwork. Sometimes the aggregation and specialization relations are thought of as separate, orthogonal representations of a semantic network. For example, it is common to have applications "walk the specialization hierarchy", or "walk the aggregation network."

Specialization and aggregation interact through *inheritance*. A specialization of an object which has aggregate parts will inherit those aggregate parts. Since the specialization relation is transitive, inheritance is also transitive. For example, if Ford-Cars is a specialization of cars, then all Ford-Cars will have wheel, door, and engine parts. Ford-Cars can, in addition, have parts defined on them that are specific only to Ford-Cars (and specializations of Ford-Cars).

The vertices of AdaKNET networks are called *Concepts*. AdaKNET has two kinds of concepts: *Individual Concepts* and *Generic Concepts*. Generic Concepts denote classes of objects, such as the Generic Concept "REUSABLE-COMPONENT" (see figure 1). Individual Concepts denote instances of these classes, such as actual code bodies stored in a library of software parts (none shown in figure 1).

Generic Concepts are arranged in the generic taxonomy i.e. the *specialization hierarchy*. All Generic Concepts participate in the specialization hierarchy. The term

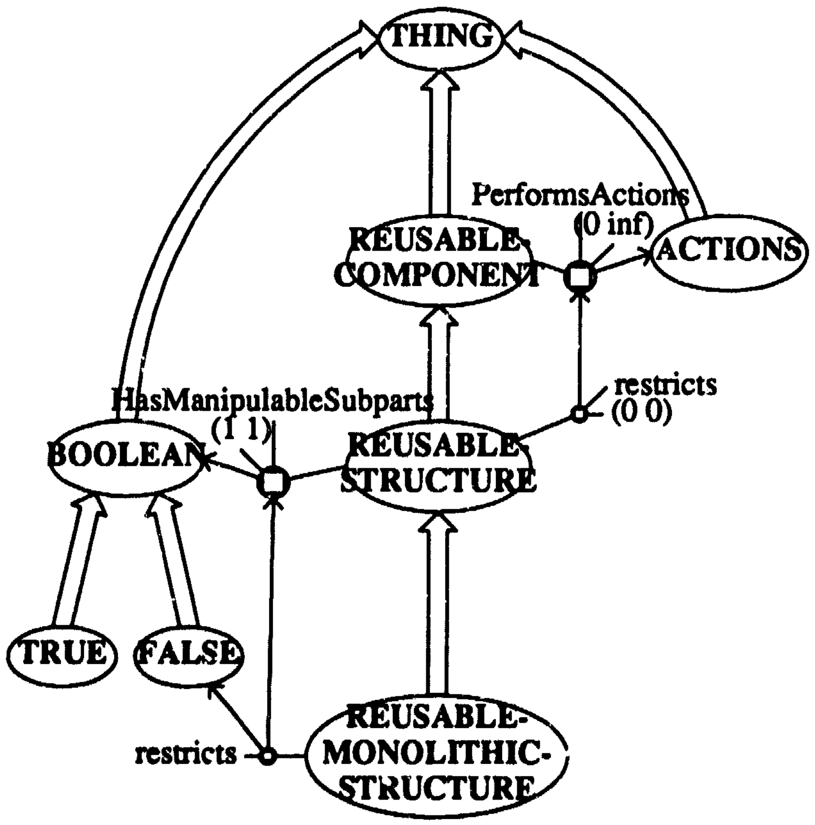

Figure 1. Library Model Fragment

"hierarchy" is used to emphasize the fact that the subnetwork induced by the specialization relations is acyclic.

Individuation is a relation which relates generic concepts to instances of generic concepts, called *Individual Concepts*. Individual Concepts may individuate one or more generic concepts. All Individual Concepts individuate at least one Generic Concept.

Aggregation is represented in AdaKNET by *Roles*. For example, in figure 1 the Generic Concept "REUSABLE-STRUCTURE" has Roles denoting properties, such as whether the reusable structure has manipulable subparts (i.e. is a polylithic data structure). Roles are inherited through specialization.

Roles convey what aggregation relations exist on Concepts; *Rolesets* describe the Role_Range and Role_Type conditions of Roles. Role_Range describes how many parts are described by an aggregate; for example, a REUSABLE-COMPONENT may perform zero or more actions of some kind. Role_Type describes what kind of part is described by an aggregate; for example, the Role_Type for the above mentioned role would be ACTIONS. AdaKNET has two kinds of Rolesets: *Generic_Rolesets* and *Particular_Rolesets*; the former describes Roles of Generic Concepts, the latter describes Roles of Individual Concepts.

Specializations of Generic Concepts inherit Roles; the properties of these inherited Roles (Role_Range, Role_Type) can be further constrained. For example, the Generic Concept REUSABLE-MONO-STRUCTURE constrains the Role_Type of the inherited Role "HasManipulableSubparts" to be FALSE. Role_Ranges can likewise be constrained. For example, the Generic Concept REUSABLE-STRUCTURE constrains the Role_Range of the inherited Role "PerformsActions" to be zero (i.e. structures do not perform actions).

Finally, Roles can be divided into *Subroles* through Roleset differentiation. Each Subrole then represents a specialized form of the Superrole; for example, a role "Children" might have two subroles "Sons" and "Daughters".

These features of AdaKNET are discussed more thoroughly in Appendix A.

3.1. AdaKNET Implementation Architecture Description

This section describes the overall AdaKNET architecture. Although from the user's perspective AdaKNET will be viewed as a single abstract data type, in reality this ADT is designed and implemented as a succession of layered abstractions, with each layer providing specialized services. This layering is illustrated in figure 2.

The AdaKNETs ADT packages several distinct object types: the top-level AdaKNET object type, and several *constituent* object types. The AdaKNET object type implements instances of semantic networks; an application may create and manipulate several semantic networks simultaneously. Each instance of AdaKNET manages collections of constituent ADTs, for example Generic Concepts, Rolesets, etc. These constituent object types, as well as some intermediary-level types manipulated by the implementation but not passed on to users of AdaKNETs, are implemented at various layers in the AdaKNET system.

The innermost layer is the set of packages which implement constituent object types of an AdaKNET network; the object layer introduces the building blocks necessary for

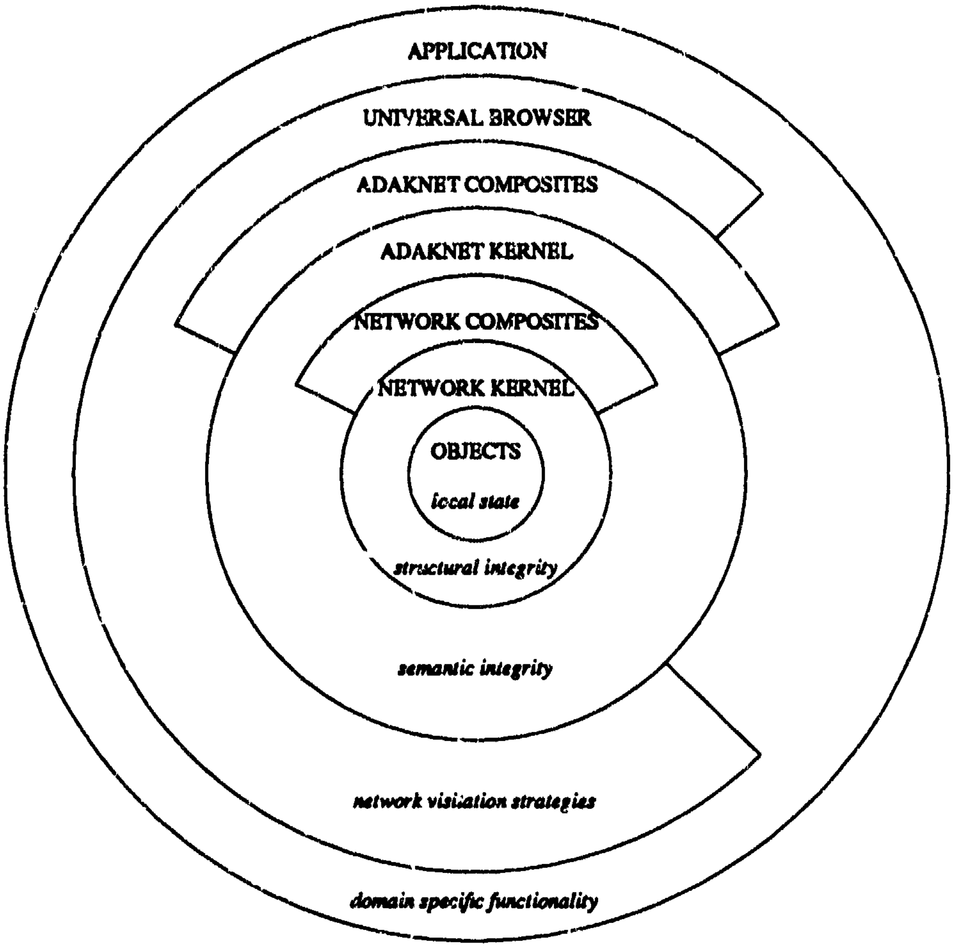

Figure 2. AdaKNET Layered Abstractions

creating a network. Roughly, each object in the conceptual level definition (e.g. concepts, roles, etc.) has a corresponding object in the object layer. Each object is implemented as an ADT, thus encapsulating the representational details of objects. Operations provided by these objects include creation, destruction, examination and manipulation of object state. Additionally, these abstractions provide primitive collection abstractions which maintain information about which object belongs to which AdaKNET instance.

The second layer is the Network package, which combines objects into a network. This layer places an abstraction wrapper around the primitive objects defined at the object layer, so that the object layer will not be visible to users of the networks layer.

Additionally, the networks layer exports the type Network used by the AdaKNET kernel. The network layer provides operations to create, modify, and examine the structure of a network. All operations at this level preserve the *structural integrity* of the network, e.g., all concepts participate in the taxonomy. The network level maintains sets of relations between objects in the object level, such as the specialization and aggregation relations. Further, the network level manages the details of saving and loading network instances.

The next layer is the AdaKNET package, which provides the user interface. The AdaKNET layer packages the structure operations of the network layer into the desired interface, and ensures that all AdaKNET networks maintain *semantic integrity*, i.e., subsumption semantics. The modification operations at this level will only make changes to a network if those changes will result in a subsumption preserving network.

An application that makes use of AdaKNET operations can interface to AdaKNET at the appropriate abstraction level (indicated by the four complete circles in figure 2). Many applications will use AdaKNET at the AdaKNET kernal level. Some composite operations as well as a network browser capability are also available to applications. These are indicated in the outermost concentric circle.

In addition, internal use of AdaKNET has often included the attachment of additional information (or *state*) to network nodes. Networks augmented with such information are called *hybrid* networks. This *AdaKNET_State* layer (not depicted in figure 2) is implemented as a generic package which parameterizes the AdaKNET abstraction to allow the association of a user-defined state type with AdaKNET constituent objects. In addition to the basic AdaKNET operations, the AdaKNET_State layer provides operations for associating and retrieving state from AdaKNET objects. For example, the Librarian instantiation of this layer associates AdaTAU inferencers with Concepts. The interested reader should see the Librarian User Manual; hybrid inferencing techniques is beyond the scope of this manual. A more detailed picture of the AdaKNET package interconnections (without the hybrid layer) is illustrated in Figure 3.

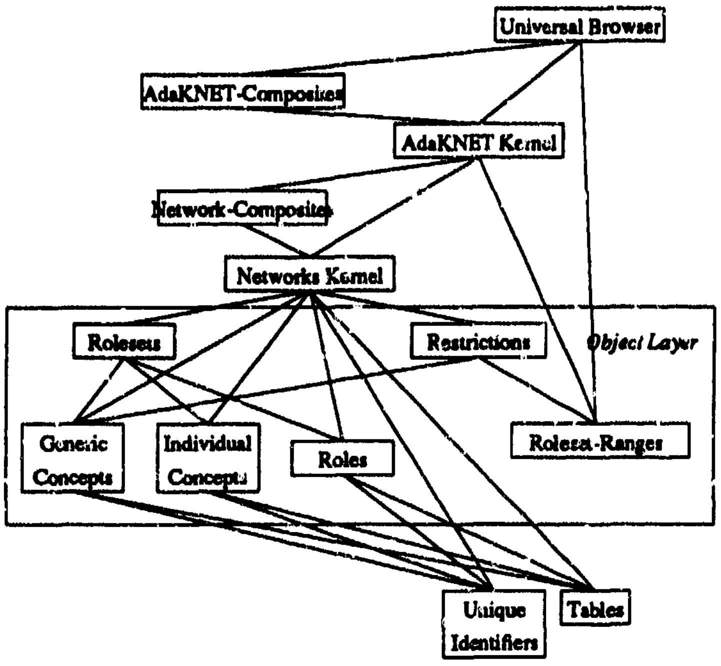

Figure 3. AdaKNET Abstraction Dependencies

4. Package AdaKNETs

This section provides an overview of the AdaKNETs package, and the relationship between this package and the AdaKNET conceptual model described in the preceding section. In the following discussion, the plural AdaKNETs will refer to the package name and ADT (e.g., the AdaKNETs ADT); the singular AdaKNET will refer to instances of AdaKNETs, as well as to the conceptual model name (i.e. AdaKNET conceptual model).

4.1. Mapping the Conceptual Model to the Implementation

One of the design goals for AdaKNET was to provide as close a match as possible between the conceptual model supported by AdaKNET and the ADT definitions used to implement AdaKNET. Our approach has been to maintain this correspondance via mappings from *conceptual objects*, i.e., those objects defined in the conceptual model, to *implementation objects*, i.e., those that appear as ADTs in the AdaKNET implementation. Figure 4 shows the relationship between implementation objects and conceptual objects.

Conceptual Objects	Implementation Objects
Generic Concept Individual Concept	Generic_Concept Individual_Concept
Role	Role
Subrole	Role[†]
Generic Roleset Particular Roleset	Generic_Roleset Particular_Roleset
Irole	[‡]

Figure 4. Conceptual/Implementation Object Mapping

Note that not every implementation object (ADT) corresponds to a conceptual object; some implementation objects simply provide useful wrappers for some information, or hide representation details of some sort. For example, the implementation object *AdaKNET* does not correspond to a conceptual object. Additionally, not every conceptual object appears as an ADT to AdaKNET application programmers; for example, *Iroles* are not implemented as ADTs, but rather as a relation. (See appendix A for more information on AdaKNET's conceptual level.)

Classes of AdaKNET Operations

AdaKNETs partitions its operations into several categories, each of which is summarized:

(1) **Construction Operations.** These operations manage the creation and destruction of AdaKNET objects. The form of these operations differs among the different object types, corresponding to constraints imposed by the AdaKNET implementation. For example, since AdaKNET enforces strict subsumption, each Generic Concept must participate in the specialization hierarchy. Therefore, no operation is provided to *create* a single Generic Concept instance; rather new instances are returned as a result of calling a composite operation which creates a Generic Concept, and links it into the AdaKNET taxonomy.

(2) **Modification Operations.** Once objects have been created, various attributes may be modified. There is a somewhat difficult distinction drawn between modification of the AdaKNET object itself, and modifications to the state of component ADTs managed by AdaKNET instances (e.g., Generic Concept ADT). For example, we view *name* to be an attribute of the Generic Concept ADT — and hence view modification of concept names as a modification of the Generic_Concept instances.

[†]A superrole / subrole relation distinguishes roles from subroles in the implementation.
[‡]Iroles are represented as 3-ary relations.

On the other hand, the links between concepts, e.g., roles, are viewed as attributes of the AdaKNET instance and not of the Generic Concept objects which participate in the aggregation hierarchy. Therefore, adding new roles or restricting existing roles are viewed as AdaKNET modification operations, not Generic_Concept modification operations. Describing some operations as AdaKNET modifiers, and other operations as constituent object modifiers represents an organizational bias to impose additional order on the somewhat complex set of AdaKNET operations; however, other organizational schemes are possible since most operations take both an AdaKNET and constituent object as parameters.

(3) **Query Operations.** These operations return information about the structure of AdaKNET instances, e.g., "what are the superconcepts of concept X?". These are the operations that will be used by applications which inference over AdaKNET instances. As with modification operations, the categorization of query operations as AdaKNET instance or constituent ADT queries reflects a package-level organizational bias. For example, although adding superconcept links to a Generic_Concept instance is considered an AdaKNET modification operation, retrieving this information is considered a Generic_Concept query.

(4) **Predicates.** These operations are boolean functions which act in two guises: as basic comparators e.g., "are these two objects the same (EQUAL) object?", and as *probes* e.g., "does object X exist in AdaKNET instance Y?". Probes can be used to preemptively test for conditions which would raise exceptions. Comparators are used to insulate application programs from distinctions between copy and share semantics for assignment. For example, AdaKNET Generic_Concepts are actually implemented as pointers to pointers to a low-level ADT implementation of Generic_Concepts; in some cases objects may be EQUAL, but not "=" (Ada predefined equality). User-defined equality hides such details.

4.2. Overview of Package AdaKNETs

This section presents a list of operations provided to create, manipulate, and examine instances of AdaKNET. These operations are partitioned into construction, modification, examination, and predicate categories. For each operation, the subprogram kind is indicated (function or procedure), with return values indicated for each function. Actual parameter profiles can be found in the AdaKNET package specification.

NOTE #1: The operations described below are found in the AdaKNETs package — this package defines the kernel operations on AdaKNET instances. An additional package, AdaKNET_Composites, provides a set of operations which are implemented in terms of kernel AdaKNETs. See section 3.1 (and the source code) for more information on these operations.

NOTE #2: The AdaKNET code uses the AdaNET spelling; we discovered the name "AdaNET" was copyrighted after a significant portion of the system had been implemented. Not all code has been revised to change references from AdaNET to AdaKNET at the time this manual is being written.

Construction Operations

The "major" objects manipulated by the AdaKNETs ADT are: AdaKNET, Generic_Concept, Individual_Concept, and Role. Construction operations are provided for each of these object types. Other objects, such as Particular_Roleset, Generic_Roleset, Roleset_Range, etc., are essentially "wrappers" for some useful information. For example, rolesets are records which pair roles and concepts, although this implementation decision is transparent to the interface. These ancillary objects are created as results of examination operations, but can be thought to have lifetimes only during execution of the program.

```
-- the following operations provide for AdaKNET persistence:
function  Create_AdaNET                         return AdaNET
function  Open_AdaNET                           return AdaNET
procedure Save_AdaNET
procedure Destroy_AdaNET
procedure Close_AdaNET

-- the following operations add information to AdaKNET instances, and
-- return constituent objects created by the operation:

procedure Add_Generic_Concept
procedure Add_Individual_Concept
procedure Add_Role
procedure Add_Partitions
procedure Add_Subsets

-- the following operations remove information from AdaKNET instances, and
-- also destroy constituent objects:

procedure Remove_Generic_Concept
procedure Remove_Individual_Concept
procedure Remove_Role
procedure Remove_Partitions
procedure Remove_Subsets
```

Modification Operations

The modification operations permit changes to object attributes, such as the *name* attribute of concepts and roles. Other attributes conceptually belong to the AdaKNET object itself, such as the relationships that exist between constituent objects, e.g., the set of specialization relations is an attribute of AdaKNET.

```
-- the following operations add semantic relations to AdaKNET instances:

procedure Add_Specialization_Link
procedure Add_Individuation_Link
procedure Add_Filler

-- the following operations remove semantic relations from AdaKNET
-- instances:

procedure Remove_Specialization_Link
procedure Remove_Individuation_Link
procedure Remove_Filler
```

```
-- the following operations modify attributes of constituent objects;
-- Rename is overloaded for concepts and roles.

procedure Rename
procedure Change_Ranges
procedure Remove_Range_Restrictions
procedure Change_Types
procedure Remove_Type_Restrictions
```

Examination Operations

Examination operations are side-effect free queries on the structure of the network. This set of operations is sufficient to support efficient inferencing on AdaKNET instances. Two flavors of examination are supported: examination of AdaKNET attributes, which yields random access to constituent objects (e.g., "return the generic concept object whose name is 'foo'"), and navigational access (e.g., "return the generic concept Y which is the superconcept of concept X").

```
-- the following operations examine attributes of AdaKNET instances, and
-- perform global queries.

function Name                        return AdaNET_Object_Name_Type
function Root_Concept                return Generic_Concept
function Generic_Concept_by_Name     return Generic_Concept
function Individual_Concept_by_Name  return Individual_Concept
function Roles_by_Name               return Role_Sets.Set

-- the following operations examine attributes of AdaKNET generic and
-- individual concepts:

function Name                        return AdaNET_Object_Name_Type
function Superconcepts               return Generic_Concept_Sets.Set
function Generic_Subconcepts         return Generic_Concept_Sets.Set
function Individual_Subconcepts      return Generic_Concept_Sets.Set
function Rolesets                    return Generic_Roleset_Sets.Set
function Rolesets                    return Particular_Roleset_Sets.Set
function Generic_Filler_Type_of      return Generic_Roleset_Sets.Set
function Particular_Filler_Type_of   return Particular_Roleset_Sets.Set
function Filler_of                   return Particular_Roleset_Sets.Set

-- the following operations examine attributes of Roles:

function Name                        return AdaNET_Object_Name_Type
function Generic_Originator          return Generic_Concept
function Individual_Originator       return Individual_Concept
function Associated_Roleset          return Generic_Concept
function Associated_Roleset          return Individual_Concept
function Superrole                   return Role
function Partition_Subroles          return Role_Sets.Set;
function Subset_Subroles             return Role_Sets.Set;

-- the following operations examine attributes of Rolesets:

function Owner                       return Generic_Concept
function Owner                       return Individual_Concept
function Associated_Role             return Role
function Range_Restriction              return Roleset_Range
function Filler_Type                    return Generic_Concept
function Fillers                        return Individual_Concept_Sets.Set
```

Predicate Operations

Predicates are boolean functions used as *probes*, i.e., to preempt the role of exceptions as a means of directing program control flow, and as simple comparators i.e., "Equal".

```
-- the following operations return Boolean results; Equal is overloaded for
-- all object types.

function Equal                         return Boolean
function Network_Exists                return Boolean
function Concept_Exists                return Boolean
function Role_Exists                   return Boolean
function Roleset_Exists                return Boolean
function Is_Range_Restricting          return Boolean
function Is_Type_Restricting           return Boolean
function Is_Range_Restricted           return Boolean
function Is_Type_Restricted            return Boolean
```

5. AdaKNET Specification Language — SNDL

SNDL is the mechanism of choice for instantiating AdaKNET knowledge bases. Although it is possible to simply write a program which instantiates AdaKNET via a sequence of calls to the AdaKNET ADT package, this mechanism is not always the most convenient (or the most descriptive, in a declarative sense) approach. We have defined a Semantic Network Description Language (SNDL) for describing AdaKNET instances in a high-level non-procedural manner.

Besides providing a convenient and readily modifiable medium for defining models, SNDL also provides services supportive of the reuse of knowledge bases as components in their own right. Since SNDL descriptions are ASCII text, AdaKNET instances described via SNDL can be transported "as is" to any site, regardless of compiler. Also, as will be seen, the language design of SNDL contains features for modularization of knowledge bases; this will be important in the reuse of knowledge bases via amalgamation of small, special-purpose knowledge bases. For example, a fragment of the Gadfly knowledge base describing Ada data types could be usefully integrated in knowledge bases of tools sensitive to Ada type semantics.

Language Goals

First and foremost, SNDL must facilitate description of AdaKNET instances. Additionally, such specifications must be easily maintainable. The former argues for a terse, concise syntax for specifying AdaKNET objects and relationships. The latter argues for sufficient "syntax" for spotlighting potentially subtle interdependencies between AdaKNET objects and relations.

The solution we have chosen attempts to make use of some of the features of Ada syntax, applied in a parallel fashion to semantic networks. The hope is that the syntax added for enhanced maintainability will not render networks difficult or clumsy to specify or examine. The following subsections describe the network structure-forming syntax. A more thorough exposition, including a complete description of the abstract syntax of SNDL, along with semantic annotations, is located in appendix B of this report.

Basic Concepts

SNDL is a language which supports definition of AdaKNET instances, and has syntactic constructs designed to highlight the mapping of the language to the AdaKNET conceptual level definition.

One important feature of SNDL which greatly augments the convenience of writing and reading SNDL specifications is that there is no need to define network structures before they are referred to. For example, a generic concept can appear as the filler type for a roleset before the generic concept is defined in the specification. In previous experience with construction of semantic network models by interactive editors, the required order of creation has proved somewhat non-intuitive (e.g., the specialization hierarchy must be built top-down, but roles and fillers must be added bottom-up). Relaxing the requirement of definition before use in the SNDL language definition should provide several advantages. Knowledge base definitions can be organized in the most easily comprehensible way for the modeler. Specifications can be modularized more easily, while preserving the overall integrity checking on the model. For example, a concept definition can include contiguous definition of the local and restricted roles of the concept; this would not be possible with a definition-before-use scheme. Later extensions to language processing tools could provide "pretty-printed" transformations of the models in various orders of presentation (e.g., depth-first, breadth-first).

The chosen form of the SNDL language provided some interesting implementation challenges. It requires a fairly complex translator implementation, since a single-pass translation will not be able to do adequate consistency checking. Here, our use of SSAGS, a Unisys-proprietary meta-generation system based on ordered attribute grammars [Payton82], is a key element to the feasibility of our approach. SNDL will evolve as experience shows what organizing schemes are most appropriate for specifying knowledge bases. A major advantage of our approach is that we are not committing to a particular organizational approach, but rather implementing a flexible specification language. Modelers will be able to use this language, not only to rapidly prototype knowledge bases, but to explore different definitional strategies as well. This methodological work is an essential prerequisite to the use of knowledge base specifications as reusable components in their own right.

Networks

AdaKNET supports the simultaneous existence of many individual knowledge bases (AdaKNET instantiations). Thus, SNDL provides a linguistic mechanism, *Network*, for encapsulating the description of an AdaKNET instance within a single language construct. The general form of AdaKNET descriptions is:

```
network Sample is
   <Semantic Object/Relation Definitions>
end Sample;
```

The SNDL *Network* construct roughly parallels the Ada *Package* construct. Both describe a named unit which encapsulates related information. SNDL also includes an *Amalgamation* construct which roughly parallels the Ada *with* construct; SNDL amalgamation supports (a limited form of) sharing of knowledge base partitions among several AdaKNET instances.

Note: Amalgamation is not implemented in the current release.

Concepts and Roles

The principal objects of AdaKNET are *Concepts*, which come in two flavors: Generic Concepts and Individual Concepts. These classes of objects are described via *concept* and *individual* structures, respectively, which are syntactic analogs of Ada record types. The *concept* and *individual* syntax indicates the position of a concept in the network via its superconcepts and provides a mechanism for encapsulating any local roles and constraints on inherited roles at the concept. The general form of object definitions is:

```
concept Sample_Concept ( <superconcepts> ) is
    <local roles and restrictions>
end concept ;

individual Sample_Individual ( <superconcepts> ) is
    <local roles and restrictions>
end individual ;
```

A concept may be distinguished from its superconcepts by restrictions on inherited roles and by local roles introduced at the concept. Such local information, as well as roleset fillers for individuals, are described as sub-structures of AdaKNET objects. A bracketing syntax is also used for these sub-structures in SNDL:

- Aggregation via *local roles...end local.*
- Restriction via *restricted roles...end restricted.*
- Differentiation via *differentiated roles...end differentiated.*
- Satisfaction via *fillers...end fillers*

SNDL Summary

SNDL provides a textual description of AdaKNET instances. A two-way translation system is provided to generate in-memory AdaKNET instances from SNDL descriptions, and to generate SNDL descriptions from in-memory AdaKNET instances. This alternative route to AdaKNET persistence (to the data structure level representation of AdaKNET instances resulting from "save" operations) provides a level of freedom for knowledge engineers to devise their own network configuration management policies, and explore reuse and amalgamation of network instances.

SNDL is designed to provide an intuitive mapping to the AdaKNET conceptual definition, and to highlight potentially subtle network semantics (e.g. roleset restriction and inheritance). This, we hope, will facilitate easier maintenance of potentially large and complex semantic networks descriptions.

A more complete definition of SNDL syntax and semantics is included in appendix B of this report.

6. Using AdaKNET

This section describes special steps that must be taken to install AdaKNET on a new host, how to create an AdaKNET knowledge base, and how to use the browser-editor application to examine and manipulate the knowledge base. Readers interested in a more in-depth discussion of semantic network inferencing (AdaKNET inferencing in particular) should consult the Gadfly user's manual, the Gadfly design report, and the AdaKNET/AdaTAU design report.

6.1. Installing AdaKNET

Detailed installation instructions are included in the Version Description Document (VDD) accompanying the source delivery of the Reusability Library Framework (RLF), including compilation order and identification of host installation dependencies. Host installation dependencies have been isolated to the *Network_Constants* package. The AdaKNET data storage model hides storage and retrieval of data files in an effort to ease later migration to relational database technology; the Network_Constants package defines the location where AdaKNET will store its data files. In the current version of AdaKNET, UNIX environment variables enable the user to specify the file system locations of RLF knowledge bases. The VDD discusses how to make use of this RLF feature.

6.2. Creating AdaKNET Knowledge Bases

AdaKNET applications require the existence of knowledge bases. These knowledge bases can be created interactively using the browser editor, they may be created programmatically as sequences of calls to the AdaKNET package, or they may be created using the SNDL processor. This section describes how to use the SNDL processor.

Use a text editor to prepare a SNDL specification. A complete syntax for SNDL is included in appendix B; however, the model illustrated in figure 5 will be used in the sample session later in this manual. Naturally, the purpose of this model is simply to familiarize you with the SNDL processor; more elaborate examples are included with the RLF delivery.

Once the specification has been prepared, it is used as input to the SNDL program generator. The output of SNDL will be a program which, when executed, will initialize an instance of AdaKNET as described in the SNDL specification. To execute SNDL on UNIX systems (assuming the file edited is called "messages.txt", and the SNDL processor is built and called "sndl") enter the following command:

```
sndl < messages.txt
```

You should get as a result the following messages:

```
Parsing input.
Parsing completed successfully.
Entering attribute evaluation phase.
Exiting attribute evaluation phase.
Entering program generation phase.
```

```
network MESSAGES is

   root concept THING is end root concept;

   concept MESSAGE (THING) is
      local roles
         SEND_DATE (1 .. 1) of DATE;
         RECEIVE_DATE (1 .. 1) of DATE;
         SENDER (1 .. 1) of PERSON;
         RECIPIENT (1 .. infinity) of PERSON;
         BODY (1 .. 1) of TEXT;
      end local;
   end concept;

   concept STARFLEET_MESSAGE (MESSAGE) is
      restricted roles
         SENDER (1 .. 1) of STARFLEET_COMMANDER;
      end restricted ;
   end concept;

   -- the following concepts are "stubs".

   concept DATE (THING) is end concept;
   concept PERSON (THING) is end concept;
   concept STARFLEET_COMMANDER (PERSON) is end concept;
   concept TEXT (THING) is end concept;

end MESSAGES;
```

Figure 5. Sample SNDL Specification

The result of this successful execution will be the generation of an Ada program named "sndlprog.a". Examination of this program will reveal that the SNDL translator has produced an AdaKNET application program to instantiate the network defined by messages.txt. Compile, link, and execute this program to instantiate the network. Note that before executing the program, the UNIX environment variable RLF_LIBRARIES must be set to the pathname of the directory that is to contain the network knowledge base (cf. the RLF Version Description Document for additional information on RLF use of environment variables). AdaKNET will initialize an AdaKNET instance, and save the instance in a directory that was established when AdaKNET was installed. The created instance can be retrieved by executing an "Open_AdaNET" operation in the AdaKNETs package; the name of the network to be opened is the name specificed in the SNDL descriptions network name. In the next section instructions on using the browser-editor to examine and manipulate the *messages* network will be provided.

SNDL Diagnostics

There are various ways SNDL specification errors can be detected. First, there are SNDL-time errors (errors caught by the SNDL processor). For example, in the *messages* description, remove the concept definition for *starfleet_commander*. Executing the SNDL command as above will produce:

```
Parsing input.
Parsing completed successfully.
Entering attribute evaluation phase.
Error: Value Restriction: starfleet_commander Undefined
Exiting attribute evaluation phase.
Entering program generation phase.
SNDL Specification errors -- no code generated
```

Other errors may not be detected by SNDL, in particular errors which deal with the subtleties of range restriction semantics. These errors will be caught at run-time when the SNDL-generated program is executed. Although SNDL will detect simple errors, it will not detect all errors. However, AdaKNET will guarantee that only subsumption-preserving networks will be instantiated.

6.3. Sample Session

In this section we provide an annotated transcript from a session using the interactive AdaKNET browser-editor. The browser-editor is an AdaKNET application delivered with AdaKNET; the Version Description Document provides instructions on building the browser-editor. In this session, we examine and modify the knowledge base created from the "message.txt" SNDL file illustrated, above. Annotations will appear in *italics*, on lines beginning with Ada-style "--" comments.

```
% browser_editor
What do you want the network name to be called?
(To abort, type '*abort*').
> MESSAGES

Positioned at--> Thing
Parents: *none*
Children:
  MESSAGE
  DATE
  PERSON
  TEXT
Individuations: *none*
```

```
Generic Rolesets: *none*

-- We wish to create a new concept to capture the notion of secret
-- communication between starfleet captains only, using specially
-- encrypted messages.

What kind of command do you want?

          1. Aggregation Network Display Commands
          2. Specialization Hierarchy Display Commands
          3. Editing Commands
          4. Move within Structure
          5. Exit the Browser

Enter number of desired command <CR>: 4

-- Commands may present a menu of other commands. In the following
-- dialogue, we will display entire menus when they are first presented;
-- thereafter we will only dislay options of interest, with ellipses "..."
-- used to show where other commands have been suppressed.

concept> STARFLEET_MESSAGE

-- The concept the browser is "focused" on is displayed after
-- each command:

Positioned at--> STARFLEET_MESSAGE
Parents:
   MESSAGE
Children: *none*
Individuations: *none*
Generic Rolesets:
   BODY(1..1) of TEXT;
   RECEIVE_DATE(1..1) of DATE;
   RECIPIENT(1..infinity) of PERSON;
   SENDER(1..1) of STARFLEET_COMMANDER;
   SEND_DATE(1..1) of DATE;


What kind of command do you want?
               ...
          3. Editing Commands
               ...

Enter number of desired command <CR>: 3

-- First we create a specialization of TEXT to capture ENCRYPTION idea:
```

```
What do you want to do?

-- The following is a list of interactive editing commands:

        1. add generic concept
        2. add individual concept
        3. add child
        4. add individuation
        5. add role
        6. add subrole
        7. restrict role
        8. remove child
        9. remove individuation
        10. remove role
        11. rename current concept
        12. rename role of this concept

Enter number of desired command <CR>: 1

What do you want the new concept to be called?
(To abort, type '*abort*').
> ENCRYPTED_TEXT
parent name> TEXT

ENCRYPTED_TEXT has been installed in the network.

Positioned at--> STARFLEET_MESSAGE
Parents:
  MESSAGE
Children: *none*
Individuations: *none*
Generic Rolesets:
  BODY(1..1) of TEXT;
  RECEIVE_DATE(1..1) of DATE;
  RECIPIENT(1..infinity) of PERSON;
  SENDER(1..1) of STARFLEET_COMMANDER;
  SEND_DATE(1..1) of DATE;


What kind of command do you want?
            ...
        3. Editing Commands
            ...

Enter number of desired command <CR>: 3

-- Now we create the specialization of STARFLEET_MESSAGE desired:
```

```
What do you want to do?
                ...
        3.  add child
                ...

Enter number of desired command <CR>: 3

What do you want the new child to be called?
(To abort, type '*abort*').
> SECRET_STARFLEET_MESSAGE
SECRET_STARFLEET_MESSAGE has been installed in the network.

Positioned at--> STARFLEET_MESSAGE
Parents:
  MESSAGE
Children:
  SECRET_STARFLEET_MESSAGE
Individuations: *none*
Generic Rolesets:
  BODY(1..1) of TEXT;
  RECEIVE_DATE(1..1) of DATE;
  RECIPIENT(1..infinity) of PERSON;
  SENDER(1..1) of STARFLEET_COMMANDER;
  SEND_DATE(1..1) of DATE;


What kind of command do you want?
```

-- Now we move to the new concept and restrict some inherited roles:

```
                ...
        4.  Move within Structure
                ...

Enter number of desired command <CR>: 4

concept> SECRET_STARFLEET_MESSAGE

Positioned at--> SECRET_STARFLEET_MESSAGE
Parents:
  STARFLEET_MESSAGE
Children: *none*
Individuations: *none*
Generic Rolesets:
  BODY(1..1) of TEXT;
  RECEIVE_DATE(1..1) of DATE;
  RECIPIENT(1..infinity) of PERSON;
  SENDER(1..1) of STARFLEET_COMMANDER;
```

```
   SEND_DATE(1..1) of DATE;

What kind of command do you want?
                  ...
            3. Editing Commands
                  ...

Enter number of desired command <CR>: 3

What do you want to do?
                  ...
            7. restrict role
                  ...

Enter number of desired command <CR>: 7

Which roleset?
```

-- *All of SECRET_STARFLEET_MESSAGE's rolesets are displayed:*

```
            1. SEND_DATE
            2. RECEIVE_DATE
            3. SENDER
            4. RECIPIENT
            5. BODY
            6. *** Abort This Menu ***

Enter number of desired command <CR>: 5

Do you want to restrict the range? n

Do you want to restrict the filler type? y

current type: TEXT
new type> ENCRYPTED_TEXT

BODY has been restricted.

Positioned at--> SECRET_STARFLEET_MESSAGE
Parents:
   STARFLEET_MESSAGE
Children: *none*
Individuations: *none*
Generic Rolesets:
   BODY(1..1) of ENCRYPTED_TEXT;
   RECEIVE_DATE(1..1) of DATE;
   RECIPIENT(1..infinity) of PERSON;
   SENDER(1..1) of STARFLEET_COMMANDER;
   SEND_DATE(1..1) of DATE;
```

-- *Note the change to the BODY role (above)*

```
What kind of command do you want?
```

-- *Restrict the RECIPIENT role to be for captains eyes only*

```
             ...
        3. Editing Commands
             ...

Enter number of desired command <CR>: 3

What do you want to do?
             ...
        7. restrict role
             ...

Enter number of desired command <CR>: 7

Which roleset?

        1. SEND_DATE
        2. RECEIVE_DATE
        3. SENDER
        4. RECIPIENT
        5. BODY
        6. *** Abort This Menu ***

Enter number of desired command <CR>: 4

Do you want to restrict the range? n

Do you want to restrict the filler type? y

current type: PERSON
new type> STARFLEET_COMMANDER

RECIPIENT has been restricted.

Positioned at--> SECRET_STARFLEET_MESSAGE
Parents:
   STARFLEET_MESSAGE
Children: *none*
Individuations: *none*
```

```
Generic Rolesets:
   BODY(1..1) of ENCRYPTED_TEXT;
   RECEIVE_DATE(1..1) of DATE;
   RECIPIENT(1..infinity) of STARFLEET_COMMANDER;
   SENDER(1..1) of STARFLEET_COMMANDER;
   SEND_DATE(1..1) of DATE;


What kind of command do you want?
                  ...
            5. Exit the Browser
                  ...

Enter number of desired command <CR>: 5

What do you want to do?


            1. Destroy this network.
            2. Exit - ignore changes from this session.
            3. Exit - save changes.
            4. Resume editing.

Enter number of desired command <CR>: 3

Exiting...
%
```

The above session shows how existing networks may be modified interactively. In general, it is far easier to use SNDL to create large, complex models, while the browser-editor is sufficient for small models like the one illustrated here.

Finally, note that a reverse translator is also provided which allows AdaKNET instances to be modified interactively, and then returned to SNDL form for archiving, revision control, etc.

APPENDIX A: Detailed Description of AdaKNET Model Semantics

AdaKNET is a knowledge representation formalism based on KL-ONE [Brachman85]. Other examples of representation systems in this family are NIKL [Kaczmarek86] and KNET [Freeman83, Searls90]. AdaKNET provides its user the ability to describe a domain by creating a model of that domain in AdaKNET, while AdaKNET's structure enforces certain consistencies between the components of that model.

Generic Concepts

In AdaKNET, the principal objects are *structured conceptual objects*, or *concepts*. AdaKNET supports two types of concepts: *generic concepts* and *individual concepts*. A generic concept models a category of things, such as the class of all humans or the class of all messages. An individual concept represents one particular thing; for instance, John Doe represents a specific human, and "my message of December 11" represents a specific message. Thus, generic concepts are roughly equivalent to "classes" in object-oriented systems, while individual concepts are roughly equivalent to "instances". For the sake of simplicity, no two concepts may have the same name.

Specialization

Generic concepts in AdaKNET are organized into a specialization hierarchy. One concept *specializes* another if the first concept represents a subset of the category described by the second concept. A sample specialization hierarchy is shown in figure 6. We see that the concept MAMMAL is defined in terms of ANIMAL, that is, MAMMAL specializes ANIMAL. Conversely, we say the concept ANIMAL *subsumes* the concept MAMMAL. The subsuming concepts are called *superconcepts* of the subsumed concepts, and the subsumed concepts are called *subconcepts* of the subsuming concepts. Because MAMMAL is directly linked to ANIMAL, we further say that ANIMAL and MAMMAL are in a *parent/child* relationship.

Specialization and subsumption are acyclic and transitive relations. So, in figure 6, HUMANs are a kind of ANIMAL, as well as a kind of MAMMAL. Specialization and subsumption are also many-to-many relations, that is, a concept may have multiple parents and children.

Individuation

Each individual concept in AdaKNET is an instance of some generic concept(s), that is, it *individuates* one or more generic concepts. Figure 7 illustrates individuation. Here JOHN-DOE individuates MAN. Individuation is preserved by subsumption, so that JOHN-DOE implicitly individuates HUMAN. In cases where it is important to distinguish between explicit and implicit individuation, we will add the term *direct* or *indirect* to the description; e.g., JOHN-DOE directly individuates MAN and indirectly individuates HUMAN.

Individuation is also a many-to-many relation; a generic concept may be directly individuated by several individual concepts, and an individual may directly individuate several generic concepts.

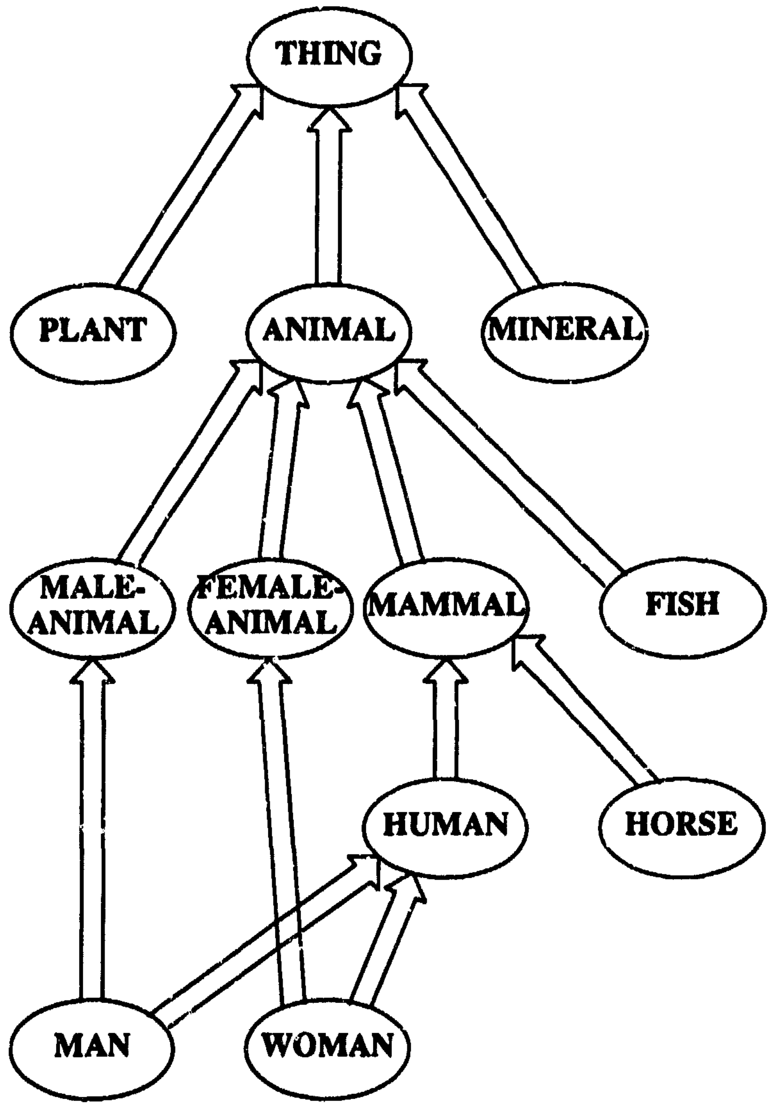

Figure 6. AdaKNET Specialization

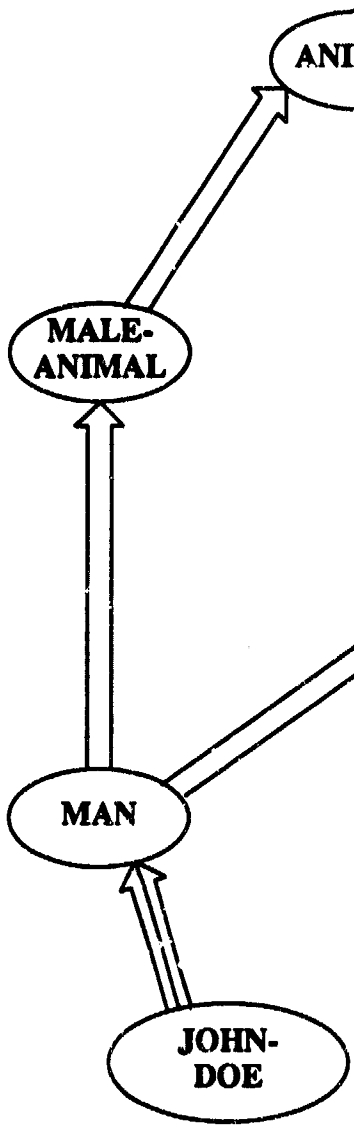

Figure 7. AdaKNET Individuation

Roles

Roles define the structure and attributes of concepts. For instance, a human has components such as a head, a torso, arms, and legs, and has attributes such as height, weight, and gender. Such attributes are represented in AdaKNET by associating roles with a concept. For example, a concept representing humans might include roles for height, weight, eye-color, etc.

Roles in AdaKNET serve either of two purposes: to indicate the general *types* of things that satisfy a given attribute or to specify the *exact* thing (the "filler", as described later) that instantiates ("satisfies", as described later) a given attribute for an individual concept. The distinction between these two is discussed in the following excerpt from [Brachman85]:

> This difference is motivated essentially by the "attributive/referential" distinction in the philosophy of language. Imagine a situation in which an alligator's tail has fallen off. We might remark, "The alligator's tail lay wriggling on the ground." Or, we might say something like, "Don't worry, the alligator's tail will grow back again." The "tails" talked about must be different in the two cases -- in the first, we are referring to the previous filler, the actual piece of protoplasm that used to be the alligator's tail. In the second, because the alligator's tail will not reattach itself to the alligator, we must mean something else by "alligator's tail." We are in fact talking in a general way about anything that will eventually play the role of "tail" for the alligator.

Rolesets are used to indicate the general, attributive flavor of roles; *Iroles* are used to indicate the specific, referential flavor of roles.

Rolesets

Rolesets are templates that identify and describe what type of thing the role's fillers should be (e.g. the height of a human is a length) and how many fillers it should have (e.g. a human has two legs). Figure 8 illustrates the notion of roleset and the associated graphical conventions. In this figure, the concept MESSAGE has five rolesets which describe the attributes all messages share. For example, every message has a date on which it was sent (corresponding to the roleset Send-Date) and a date on which it was received (corresponding to the roleset Receive-Date).

The *type* or *value restriction* of a role's fillers is specified by a generic concept associated with the roleset. In figure 8, the roleset Sender has type PERSON, indicating that senders of messages must be persons. Individuals which fill the sender roleset must therefore be individuals of type person, or be individuals of some subconcepts (directly or indirectly) of type person.

The cardinality of role fillers is specified by a roleset's *range restriction* (or, simply *range*). A range restriction consists of a lower and an upper bound on the number of fillers the role is allowed. If the lower and upper bounds of a roleset range are equal, we say the role has been *converged*. *INFINITY* as an upper bound indicates that an unlimited number of fillers are possible. A message, as defined in figure 8, has exactly one "send date" (the Send-Date role has been converged to "1"), while a message may have one or more recipients.

AdaKNET distinguishes between two kinds of rolesets: *generic rolesets* and *particular rolesets*. Generic rolesets are owned by generic concepts, while particular rolesets are owned by individual concepts.

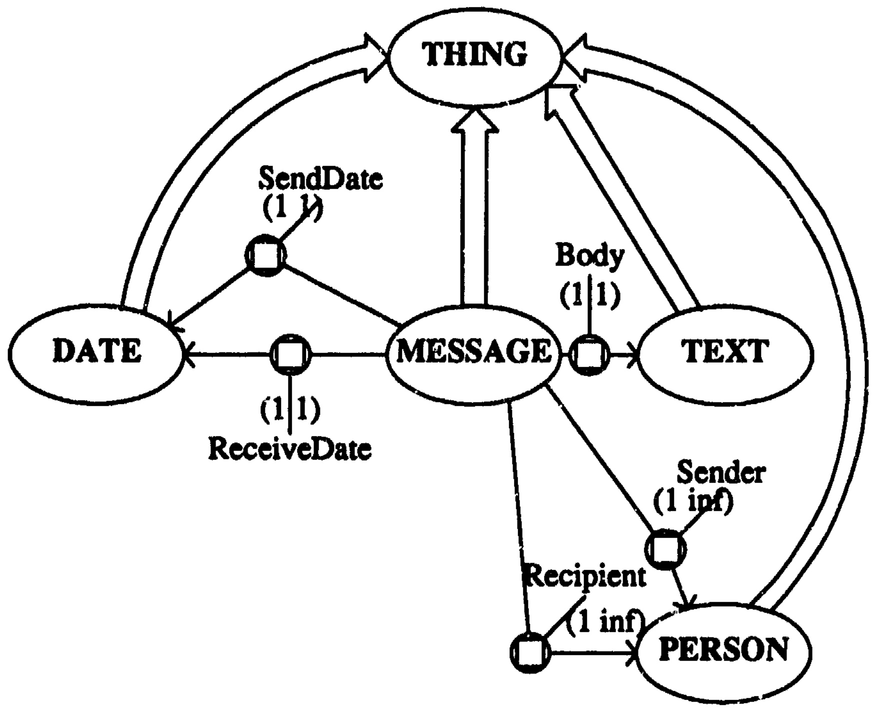

Figure 8. AdaKNET Roles

Iroles

Iroles are used to show the specific fillers of a roleset. Iroles represent tertiary relationships between the irole owner, the irole filler, and the particular roleset being filled. Because AdaKNET only allows particular rolesets to be filled, both owner and filler must be individual concepts.

Iroles are depicted by solid boxes, with links connecting the owning individual, the filler, and the satisfied particular roleset, as illustrated in Figure 9. In this example, John Doe, an individuation of Person, satisfies the Recipient role of the individual, MESSAGE-1. In AdaKNET, an individual may satisfy many roles (i.e. can participate in more that one Irole).

The filler of an individual's role must adhere to the roleset's restrictions; the irole specifying this filler is then said to *satisfy* the roleset. The satisfaction criteria are:

(1) The irole's filler must be an individual concept that individuates the type of the roleset (either directly or indirectly).

(2) For any individual concept, the number of iroles which satisfy a roleset cannot exceed the upper bound of the roleset's range.

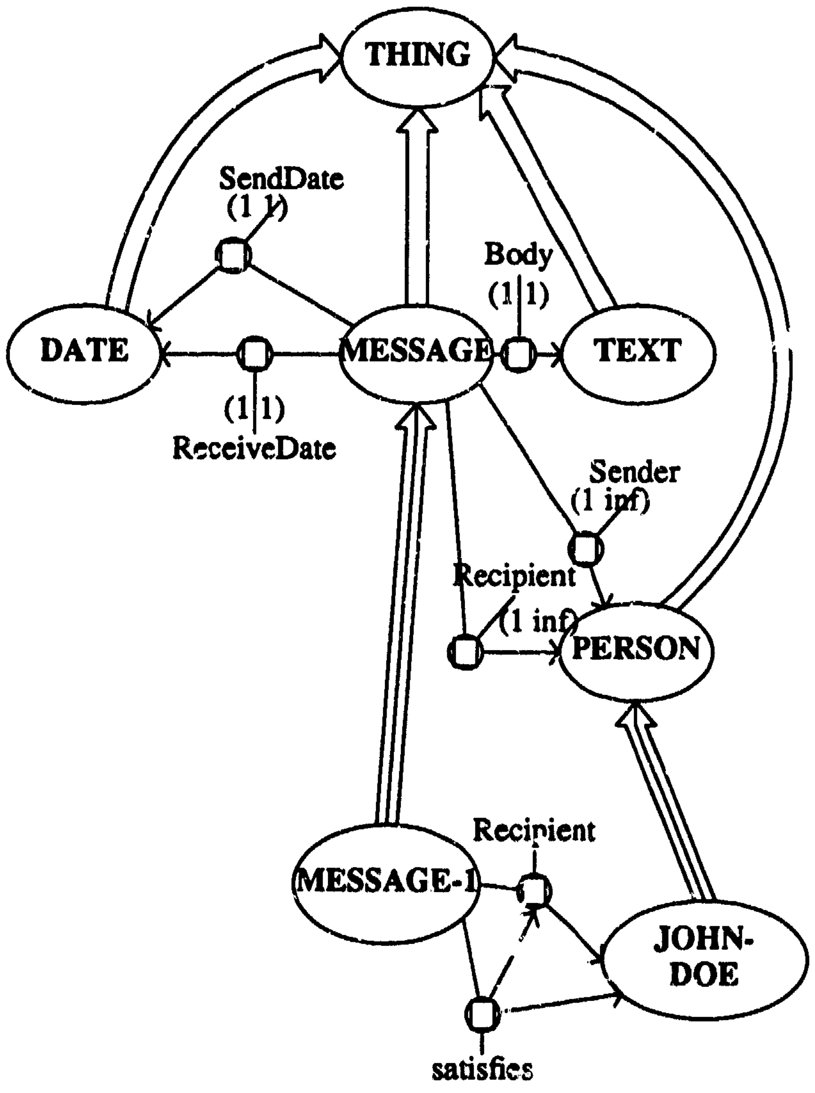

Figure 9. AdaKNET I-Roles

Inheritance

In AdaKNET, a subsumed concept *inherits* the roles of its superconcepts; that is, each role of the superconcept is also a role of the subconcept. Herein lies the power of specialization: to define a generic concept, one only needs to specify a concept's parents and that information which distinguishes the concept from its parents. Such distinguishing information may be new roles introduced at the subconcept (*locally-defined roles*), or further restrictions or differentiations on roles that are inherited. (Differentiations and restrictions are discussed in a later section.) The semantics of the subsumption relation is, essentially, that any individual of the subsumed concept is also an individual of the subsuming concept. For this to be true, subsuming concepts can only strengthen the restrictions of inherited roles. This notion of *subsumption preserving semantics* is central to understanding what constitutes legal AdaKNET models.

An individual concept also inherits the generic rolesets of the generic concepts it individuates; these generic rolesets are inherited as particular rolesets. This is one of two ways particular rolesets can be introduced at individual concepts (differentiation is the other way). In fact, this is the only way that particular rolesets are created. No new roles may be introduced at an individual concept; all roles must correspond to a role of one of the subsuming generic concepts. As with specialization, further restrictions or differentiations may be put on inherited rolesets.

Multiple Inheritance

AdaKNET allows a generic concept to specialize more than one superconcept and an individual concept to individuate more than one generic concept. This allows a concept to inherit the roles of all of its parents (specialized concepts or individuated concepts). When the parents have non-overlapping sets of roles, multiple inheritance works in the same way as single inheritance. In Figure 10, TOP-SECRET-MESSAGE inherits the role Key from ENCRYPTED-MESSAGE and the role Network from NETWORK-MESSAGE in the normal fashion. (Note that TOP-SECRET-MESSAGE further restricts the type of the role Network.)

If some parents share a role which descends from a common ancestor (i.e. there exists a single concept which subsumes the parents and from which the parents inherit the role), the role is inherited with the conjunction of the parents' restrictions on the role. The role's range must be the largest possible range that falls within all the parents' ranges for the role. TOP-SECRET-MESSAGE's inheritance of Recipient from all three of its parents illustrates this; the conjunction of the parent ranges for this role is (1,1). Similarly, the role's type must be the same as or subsumed by all of the parents' types for the role. Thus, Sender has SECRET-AGENT as its type in TOP-SECRET-MESSAGE. If a range or type meeting these criteria does not exist, the inheritance is not possible without violating subsumption, and the specialization is not allowed.

Any parent roles which have the same name but do not descend from a common ancestor are distinct roles. In order for these roles to be inherited by a single concept, the name conflict must be resolved by renaming. For example, if NETWORK-MESSAGE's role NetworkMethod were named Method, its name would conflict with ENCRYPTED-MESSAGE's role Method. In this case, TOP-SECRET-MESSAGE would not be able to be a child of both NETWORK-MESSAGE and ENCRYPTED-MESSAGE until one of

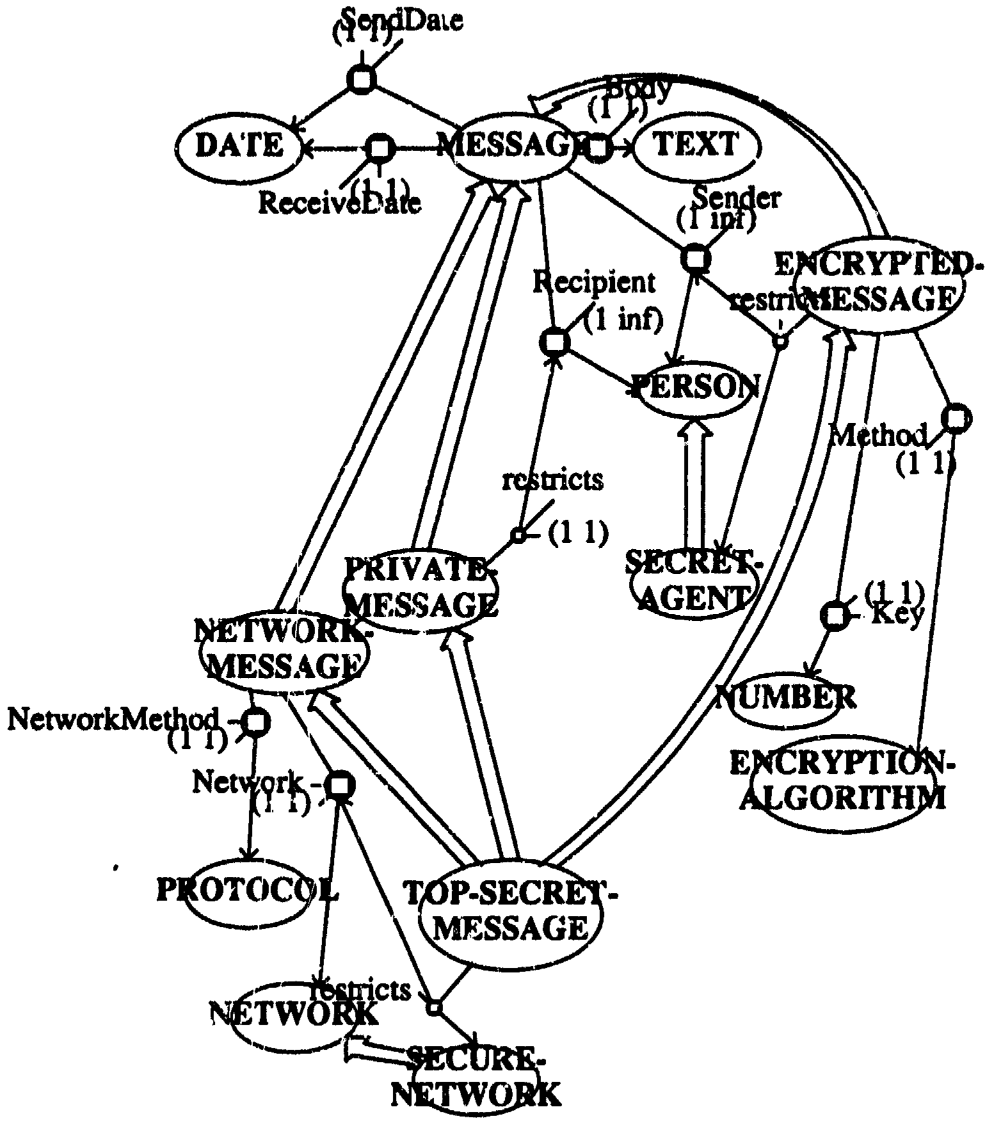

Figure 10. AdaKNET Multiple Inheritance

these roles was given a different name.

Roleset Restriction

Roleset restriction is a mechanism whereby a concept constrains the range and/or type of inherited rolesets. All roleset restrictions must preserve the semantics of the specialization and individuation relations. Since types and ranges constitute *necessary* conditions on fillers, (e.g. each MESSAGE must have at least one Sender, and each Sender must be a PERSON), this means that these conditions may not be weakened by roleset restriction. Thus, one may restrict an inherited roleset's range to be a smaller interval than the range of the parent's roleset, and/or one may restrict an inherited roleset's type to be some specialization of the type of the parent's roleset.

Figure 11 illustrates roleset restriction. The range of the role Recipient is converged to "1" for the concept PRIVATE-MESSAGE. A PRIVATE-MESSAGE is thus defined as a MESSAGE with exactly one Recipient.

Roleset restriction is denoted via the *restricts* relation. Note that the restricts relation does not introduce a new role, but rather tightens the range or narrows the type of an inherited role.

Roleset Differentiation

Roleset differentiation is denoted via the *differentiates* relation. *Roleset differentiation* allows a role to be described in a more detailed way than is possible with a single roleset. Consider the example in figure 12. One of the properties of a mail message is that it must be received by someone. This is modeled by having a roleset Recipient with type PERSON and owner MESSAGE. We may wish to make finer distinctions; for example, we may want to show that a recipient can be a primary recipient or can be a "carbon-copy" recipient. Using differentiation, we can do this by creating the subroles Primary-Recipient and CC-Recipient. The rolesets describing these subroles may have their own types and ranges to further restrict the kind and cardinality of fillers for the subroles. Thus, differentiation allows one to categorize role fillers, and to apply additional restrictions on fillers in those categories.

AdaKNET supports two classes of roleset differentiation - *partitioning* and *subsetting*. In the first, the immediate differentiators of a roleset partition that roleset, i.e., every filler of the differentiated role is a filler of exactly one of the subroles indicated by the differentiators. In our example, differentiating using partitioning implies that every recipient is either a carbon copy recipient or a primary recipient.

The second class of roleset differentiation, subsetting, is less restrictive than partitioning, allowing one to create subroles that do not fully cover all fillers of the differentiated role. If our example was created using this subsetting class of differentiation, we could have a filler of Recipient that is not a filler of either CC-Recipient or Primary-Recipient. Note that since an individual concept can participate in many Iroles, an individual can be used as a filler of more than one subrole (as well as the differentiated role itself, in the case of subsetting).

Range checking differs between the two styles of differentiation. In both schemes, the sum of any subrole's upper range bound and the other subroles' lower range bounds must not exceed the upper range bound of the differentiated roleset. This is because it is impossible to not exceed the differentiated roleset's upper range bound while having the maximum number of fillers for such a subrole and adhering to the range restrictions of the other subroles. Partitioning also requires that the sum of any subrole's lower range bound and the other subroles' upper range bounds not be less than the lower range bound of the differentiated roleset. Otherwise it is impossible to cover the differentiated roleset

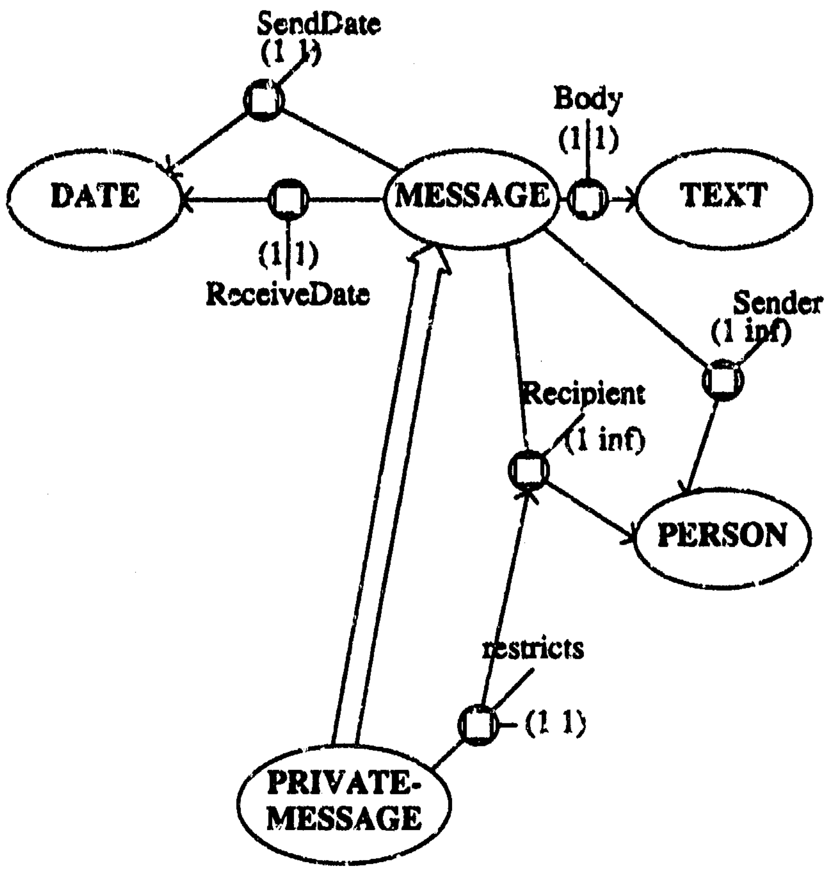

Figure 11. AdaKNET Restriction

while having the minimum number of fillers for such a subrole and adhering to the range restrictions of the other subroles. The subset class of differentiation does not impose this last restriction, because the differentiated role can have fillers that are not fillers of the differentiating subroles.

Finally, if a role is differentiated, the entire differentiation is inherited; that is, a specialization or individuation inherits the differentiated role, the subroles, and the differentiation relation among them. Because of this, it is not possible to differentiate a roleset which has been differentiated with partitioning at a subsuming concept. In the example of figure 12, for instance, the roleset Recipient cannot be differentiated again at a concept subsumed by MESSAGE if partitioning was used. Using differentiation with subsetting, further subsetting of the differentiated role is allowed at subsumed concepts. In figure 12, this would denote differentiation of the set of those fillers that do not satisfy one of the existing subroles, Primary-Recipient and CC-Recipient. With either subsetting or partitioning, we may of course differentiate one of the subroles and, for example, create subroles of the subrole Primary-Recipient.

Summary

AdaKNET is a system for representing knowledge. An AdaKNET network is a hierarchy of concepts. The concepts represent things and kinds of things and the hierarchy represents a taxonomy for these things. The attributes of concepts are modeled by the roles of a concept. AdaKNET allows one to describe both the types of things that can fill a role and the actual fillers themselves. Roles are implicitly passed down the links in the concept hierarchy. Thus a concept is totally defined by its position in the hierarchy

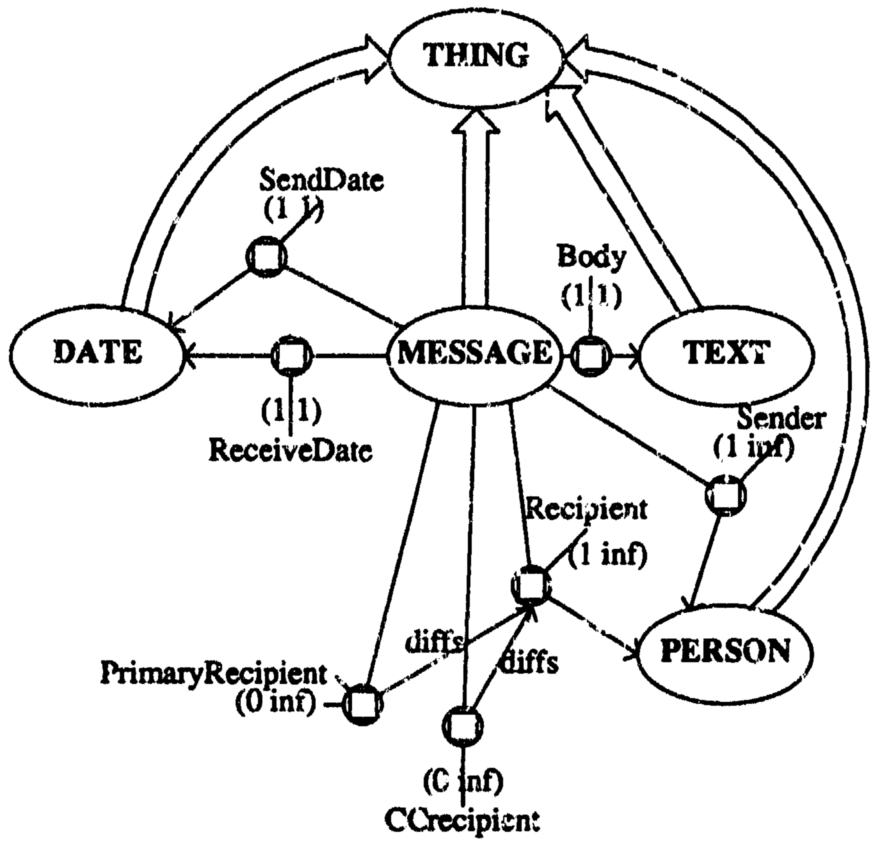

Figure 12. AdaKNET Differentiation

and the characteristics which distinguish it from the concepts directly above it. These simple objects and relations combine to form a powerful knowledge representation system.

APPENDIX B: SNDL Syntax and Summary

This appendix contains an overview of the extended BNF variant used to describe SNDL, followed by a description of the individual language features of SNDL. Each language feature is presented syntactically, with the syntactic description followed by a short summary of the semantics of the feature. Following the description of the individual features, the appendix closes with a complete syntax summary.

Extended BNF (EBNF) Meta-Symbols

The syntax of the language is described using an extended BNF. The notation used is the same as the notation used throughout the Ada LRM. A brief description is given below. For a complete description see section 1.5 of the LRM.

`lower_case_word`
: nonterminal (e.g. `adaknet_spec`).

italicized_part`_lower_case_word`
: refers to same nonterminal as the lower case word without italicized part. The italicized part is used to convey some semantic information.(e.g. *generic_concept*`_identifier`).

`bold_face_word`
: language token (e.g. **`begin`**).

`{item}`
: braces enclose item which may be repeated zero or more times.

`[item]`
: brackets enclose optional item.

`item1 | item2`
: alternation; either item1 or item2

SNDL EBNF and Semantics

AdaKNET Specifications

```
adaknet_spec ::=
    [amalgamations]
    network network_identifier is
        root_concept
        concept {concept}
    end network_identifier;

amalgamations ::= with network_identifier {, network_identifier};
```

An AdaKNET specification consists of exactly one network definition; the identifier denoting this network must match at the end keyword.

The networks identified as amalgamations must have as their root concept a generic concept which exists in the current network specification. The interpretation of amalgamations is that the imported network rooted at concept C replaces the concept of the same name in the current network.

Root Concept

```
root_concept ::=
     root concept generic_concept_identifier is
        [local_roles]
        [differentiated_roles]
     end root concept;
```

Each network must have one distinguishing generic concept which subsumes all concepts in the network, and is itself subsumed by no concepts. Since this *root concept* does not inherit roles, only local roles can be specified (via local role definition and differentiation of these local roles).

Concept Definitions

```
concept ::= generic_concept | individual_concept
```

Network definitions consist of definitions of concepts, and the relationship between concepts. There are two types of concepts in AdaKNET: individual concepts, and generic concepts (see Appendix A).

Generic Concept

```
generic_concept ::=
     concept generic_concept_identifier ( specializes ) is
          [local_roles]
          [restricted_roles]
          [differentiated_roles]
     end concept;

specializes ::=
     generic_concept_identifier {, generic_concept_identifier}
```

All generic concepts, except the root concept, must specialize at least one other generic concept.

Individual Concept

```
individual_concept ::=
     individual individual_concept_identifier ( individuates ) is
          [restricted_roles]
          [differentiated_roles]
          [satisfied_roles]
     end individual;

individuates ::=
     generic_concept_identifier {, generic_concept_identifier}
```

All individual concepts must individuate at least one generic concept.

Roles

This syntax defines how roles are introduced, and how inter-role relationships (i.e. restriction, differentiation, satisfaction) are specified.

Local Roles

```
local_roles ::=
      local roles
          role {role}
      end local;

role ::=
      role_identifier (number .. number_or_infinity)
          of generic_concept_identifier;
```

Roles introduced in the **local roles ... end local** section are considered to introduce new roles into the network (not constrain existing roles).

Restricted Roles

```
restricted_roles ::=
      restricted roles
          restriction {restriction}
      end restricted;

restriction ::= range_restriction | value_restriction |
      range_and_value_restriction

range_restriction ::= role_identifier (number .. number_or_infini

value_restriction ::= role_identifier of generic_concept_identifier;

range_and_value_restriction ::=
      role_identifier (number .. number_or_infinity)
          of generic_concept_identifier;
```

Restrictions in the **restricted roles ... end restricted** section are considered to restrict the satisfaction conditions on inherited roles. Therefore, the *role_*identifier must correspond to an inherited role. The restrictions must be consistent with the inherited conditions as discussed in Appendix A.

Differentiated Roles

```
differentiated_roles ::=
    differentiated roles
        differentiation {differentiation}
    end differentiated;

differentiation ::=  subset | partition
```

```
subset ::=
    subset role_identifier into
        role {role}
    end subset;

partition ::=
    partition role_identifier into
        role {role}
    end partition;
```

Differentiators may only differentiate existing roles; these roles may be local or inherited. Each differentiation consists of a declaration of a set of roles. These roles are considered to be local roles of the concept where the differentiation is introduced. The semantics of subset and partition differentiation is discussed in Appendix A.

Satisfied Roles

```
satisfied_roles ::=
    fillers
        filler {filler}
    end fillers;

filler ::= individual_concept_identifier satisfies role_identifier;
```

For each *filler*, the *individual_concept_*identifier must refer to an individual defined in the network, and the *role_*identifier must correspond to a particular role that is either inherited or locally introduced via differentiation. Fillers must adhere to the restrictions of the roleset they satisfy, as discussed in Appendix A.

Lexical Elements

```
identifier ::= letter {[underline] letter_or_digit}

letter ::= upper_case_letter | lower_case_letter

number ::= digit {digit}

number_or_infinity ::= number | infinity

string ::= "{graphic_character}"
```

SNDL EBNF Syntax Summary

The following is the EBNF description of the SNDL syntax. Terms are introduced in depth-first fashion.

```
adaknet_spec ::=
    [amalgamations]
    network network_identifier is
        root_concept
        concept {concept}
    end [network_identifier];

amalgamations ::= with network_identifier {, network_identifier};

root_concept ::=
      root concept generic_concept_identifier is
         [local_roles]
         [differentiated_roles]
      end root concept;

concept ::=  generic_concept | individual_concept

generic_concept ::=
      concept generic_concept_identifier ( specializes ) is
           [local_roles]
           [restricted_roles]
           [differentiated_roles]
      end concept;

specializes ::=
      generic_concept_identifier {, generic_concept_identifier};

individual_concept ::=
      individual individual_concept_identifier ( individuates ) is
           [restricted_roles]
           [differentiated_roles]
           [satisfied_roles]
      end individual;

individuates ::=
       generic_concept_identifier {, generic_concept_identifier};

local_roles ::=
       local roles
           role {role}
       end local;

role ::=
       role_identifier (number .. number_or_infinity)
           of generic_concept_identifier;

restricted_roles ::=
       restricted roles
           restriction {restriction}
       end restricted;
```

```
restriction ::=
      range_restriction | value_restriction |
      range_and_value_restriction

range_restriction ::= role_identifier (number .. number_or_infini

value_restriction ::= role_identifier of generic_concept_identifier;

range_and_value_restriction ::=
      role_identifier (number .. number_or_infinity)
          of generic_concept_identifier;

differentiated_roles ::=
      differentiated roles
          differentiation {differentiation}
      end differentiated;

differentiation ::=  subset | partition

subset ::=
      subset role_identifier into
          role {role}
      end subset;

partition ::=
      partition role_identifier into
          role {role}
      end partition;

satisfied_roles ::=
      fillers :
          filler {filler}
      end fillers;

filler ::= individual_concept_identifier satisfies role_identifier;

identifier ::= letter {[underline] letter_or_digit}

letter ::= upper_case_letter | lower_case_letter

number ::= digit {digit}

number_or_infinity ::= number | infinity

string ::= "{graphic_character}"
```

References

[Barr81]	A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence, Volume 1*, William Kaufmann, Inc., 1981.
[Booch87]	G. Booch, *Software Components with Ada*, Benjamin/Cummings Publishing Company Inc, Menlo Park, California, 1987.
[Brachman85]	R. J. Brachman and J. Schmolze, "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, 9(2) (Spring 1985), pp. 171-216.
[Freeman83]	M. W. Freeman, L. Hirschman, D. P. McKay, F. L. Miller, and D. P. Sidhu, "Logic Programming Applied to Knowledge-Based Systems, Modelling, and Simulation," *Proceedings of the Conference on Artificial Intelligence*, April 1983, pp. 177-193.
[Kaczmarek86]	T. S. Kaczmarek, R. Bates, and G. Robins, "Recent Developments in NIKL," *Proceedings AAAI-86*, Philadelphia, PA, August 1986, pp. 978-985. Fifth National Conference on Artificial Intelligence.
[LRM83]	*Reference Manual for the Ada Programming Language*, United States Department of Defense, February 1983. (American National Standards Institute/MIL-STD-1815A-1983).
[McDowell89]	R. McDowell and K. Cassell, "The RLF Librarian: A Reusability Librarian Based on Cooperating Knowledge-Based Systems," *Proceedings of RADC 4th Annual Knowledge-Based Software Assistant Conference*, Utica, NY, September 1989.
[Payton82]	T. F. Payton, S. E. Keller, J. A. Perkins, S. Rowan, and S. P. Mardinly, "SSAGS: A Syntax and Semantics Analysis and Generation System," *Proceedings of COMPSAC '82*, 1982, pp. 424-433.
[Searls90]	D. B. Searls and L. M. Norton, "Logic-Based Configuration with a Semantic Network," *Journal of Logic Programming*, 8(1,2) (1990), pp. 53-73.
[Simos88]	M. Simos, "The Growing of an Organon: A Hybrid Knowledge-Based Technology and Methodology for Software Reuse," *Proceedings of 1988 National Institute for Software Quality and Productivity (NISQP) Conference on Software Reusability*, April 1988, pp. E-1 through E-25.
[Smith77]	J. M. Smith and D. C. P. Smith, "Data Abstraction: Aggregation and Generalization," *ACM Transactions on Database Systems*, 2(2) (June 1977), pp. 105-133.
[Solderitsch89]	J. Solderitsch, K. Wallnau, and J. Thalhamer, "Constructing Domain-Specific Ada Reuse Libraries," *Proceedings of Seventh Annual National Conference on Ada Technology*, March 1989.
[Wallnau88]	K. Wallnau, J. Solderitsch, M. Simos, R. McDowell, K. Cassell, and D. Campbell, "Construction of Knowledge-Based Components and Applications in Ada," *Proceedings of AIDA-88, Fourth Annual Conference on Artificial Intelligence & Ada*, November 1988, pp. 3-1 through 3-21.