European Strategic Programme for Research

ESPRIT

and Development in Information Technology

ESPRIT Project 813
TODOS

Architecture Specification Language:
Design and Implementation

D. Castelli, C. Meghini, D. Musto

Technical Report on Activity A4.2

May 1988

Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

Architecture Specification Language: Design and Implementation (*)

D. Castelli, C. Meghini, D. Musto

Consiglio Nazionale delle Ricerche
Istituto di Elaborazione della Informazione
Via Santa Maria, 46
I-56100 Pisa, Italy

(*) A short version of this paper has been accepted at the IFIP WG 8.4 Working Conference on 'OFFICE INFORMATION SYSTEMS: The Design Process', to be held in Linz, Austria, 15-17 August, 1988.

Abstract

The report presents the design and the implementation of ASL, a Language for specifying Office Information System Architectures. An Office Information System Architecture is seen as a set of interconnected hardware components, running software packages that perform typical office activities. The Language presented adopts the object oriented representation paradigm, and organizes the specification of an Architecture through four level of abstractions. The knowledge on commercial hardware and software components that are employed in architectures is collected in the ASL Catalogue. ASL is implemented on top of PSN, an extension of Lisp with knowledge structuring facilities. Data structures and programs that handle architectures are then embodied in a PSN knowledge base and manipulated by the PSN interpreter. The interface provided with the Language uses windows, menus and Sunwindow text editing capabilities to facilitate the specification of ASL operations and the visualization of their results.

1. Non Technical Summary

The present report is part of Activity T.4.2, Architecture Specification Language, of Work Package 4 (WP4). It presents the results of Activity A4.2.4, Implementation of ASL Knowledge Base. In the Project Workplan, there is no provision for this report. However, we think that the separation of the development of ASL from the development of the simulation tool makes it necessary. Moreover, it has been thought that this report might be a helpful companion of the demonstration of the ASL prototypical implementation.

1.1. Contents

In Section 3 the concepts that have guided the Language design are described, along with an introduction to the implementation framework. In Section 4 we present the operations that can be used to manipulate the various components of architectures. Section 5 introduces the Language implementation, while Section 6 presents a description of the ASL User Interface. In Section 7, a sample insertion in the ASL Catalogue is given, as an example of the use of ASL. In Appendix A the complete is-a hierarchy of ASL is shown, whereas in Appendix B a complete description of the ASL Interface operations is presented.

1.2. Relation to Previous Work

The work reported in this document is the complement of Activities A.4.2.1 (Architecture Specification Language Design), and A.4.2.2 (Definition of ASL Knowledge Base), on the development of a language for representing architectures that support office information systems.

1.3. Relation to Future Work

The Architecture Specification Language will be the target language of Activity A.4.3.3 (Mapping Techniques from Conceptual Models to Architecture Models) and will serve as a basis for A.4.3.5 (Architecture Simulation). The former Activity concerns the transformation of a TCL Conceptual Schema into a set of ASL Architectures; the latter deals with the simulation of office activities on an ASL Architecture.

1.4. Relation to Work in Other Work Packages

The description of the equipment employed in the Office is part of the information that must be collected by Work Package 1 (WP1), in the Requirements Collection and Analysis Phase. The Architecture Specification Language could be usefully adopted for such description. To this end, there have been contacts between WP1 and WP4, and a Meeting in Milan, in which the model underlying ASL has been illustrated to ITALTEL.

1.5. Effort

The preparation of this report has required about 60 man days.

2. Introduction

The design of an Office Information System is seen in the TODOS Methodology [Pern86] as consisting of four strongly integrated phases: requirements collection and analysis, logical design, rapid prototyping and architecture design.

While the first three phases are common to any traditional software development methodology, the architecture design phase is a contribution of the TODOS Project to the area of office information system design. The aim of the architecture design phase is the identification of an architecture that realizes the office information system being designed. By architecture any set of interconnected hardware components, running software packages, is meant. Thus, rather than implementing the office information system by developing software, the TODOS methodology focuses on the use of existing software packages and emphasizes the design of the architecture that will support the information system.

To achieve its aim, the architecture design phase is subdivided into two stages. In the first stage, which we call the architecture generation stage, a number of alternative architectures, suitable for realizing the office information system, is identified. The input to the architecture generation is quantitative and qualitative information about the office activities. The qualitative information is provided by the logical design phase, in the form of a conceptual schema describing the data objects manipulated by the office activities and the functions abstracting such activities [Barb87]. The quantitative information is provided by the requirements collection and analysis phase [Bass87], and integrates the conceptual schema with data such as the frequency of activities, and the size of the involved data. Given these two kinds of information, architectures are generated by the interaction of the office system designer with the architecture generation tool, which assists the designer in the specification of architectures satisfying the requirements given as input.

In the second stage of the architecture design phase, called architecture selection stage, the most appropriate architecture, among those identified in the first stage, is selected. To this end, each candidate architecture is transformed into an equivalent queuing network model, on which transactions simulating the office activities are run. The performance of the architecture is so evaluated, so that the most appropriate architecture can be selected on the basis of a cost-benefit tradeoff.

In [Cast88] a Language for Architecture Specification (ASL) that can be usefully employed within the architecture generation phase of the TODOS methodology has been introduced. In the present report, a refinement of the Language design and an implementation of the Language is described. Such implementation consists of a (prototypical) language interpreter and user interface. The ASL interpreter is built on top of the interpreter of PSN, an extension of Franz Lisp with knowledge structuring facilities. The user interface is written on top of the Sunwindow software package; it uses windows, menus and graphics to facilitate the interaction of the user with ASL. The prototypical ASL implementation is being developed on a SUN3 Workstation, running Unix 4.2, release 3.2.

The report is organized as follows: in the next Section the principles of ASL will be outlined, while in Section 4 a high level description of the operations available for manipulating architectural objects will be presented. Section 5 describes the knowledge base supporting the language, showing how the relevant knowledge on hardware and software components, and on complex combinations of them, has been structured. The ASL Interface is introduced in Section 6, while an example of the use of ASL is given in Section 7. In Appendix A the complete is-a hierarchy of ASL is shown, whereas in Appendix B a description of the ASL Interface operations is presented.

3. Architecture Specification Language: Ideas and Concepts.

The Architecture Specification Language (ASL) is a language for representing computer system architectures of office information systems. Such architectures consist of interconnected hardware components, supporting the functionalities of software components.

A hardware component is any physical device that can be employed in an information system; computers, input/output peripherals, local area networks are typical hardware components. Software components are the software packages that run on the computers of architectures, performing tasks that may involve other hardware components.

The connections among hardware components may point-to-point or multipoint. A point-to-point connection is a physical link between two hardware components that allows the communication between them. A multipoint connection enables the communication among several hardware components, so establishing a computer network. Within the variety of computer networks that have been proposed [Stal84], ASL only considers Local Area Networks (LAN's), as they are typically employed in Office Information Systems.

Architectures, in their most general form, are modelled by ASL as consisting of LANs which communicate between each other. A LAN, in turn, is seen as a set of Subsystems, where a Subsystem represents a local host in a computer network. Finally, a Subsystem is constituted by a computer Unit, point-to-point connected to other hardware Units. This view determines a top-down decomposition of Architectures through four levels:

(1) Architecture level, where Architectures are defined in terms of LAN's;

(2) Local Area Network level, which we will simply call Network level, where LAN's are defined as sets of interconnected Subsystems;

(3) Subsystem level, including the definition of Subsystems as sets of point-to-point connected hardware Units; and

(4) Unit level, the lowest level, where Units are defined.

Units are thus simplest constituents of Architectures and refer directly to commercial components. In fact, a Unit represents a particular hardware or software component of an Architecture. The relationship between commercial components and Units resembles that between a prototype and its instances [Sowa84], and is clearly one-to-many: many instances of the same component may be employed in an Architecture; each one of these instances will be represented by a Unit, but all these Units will refer to the same component.

The hardware and software components referenced by Units form the Catalogue of the ASL. The ASL Catalogue may be thought of as staying at the fifth level of the Architecture decomposition shown above, in the sense that the objects that it contains are used to define Units. In a more general sense, however, the relationship between Units and Catalogue objects is different from the relation between the other levels of the decomposition, thus we have preferred to keep the Catalogue as a separate concept. From a more formal point of view, the Catalogue can be seen as the domain of ASL, containing the language constants used by higher level constructors to build the language objects. Thus the role of the Catalogue in the ASL is analogous to that played by numbers and strings in traditional data modelling languages. Of course, ordinary constants like numbers and strings are also available in ASL.

At the knowledge level, ASL can then be seen as embodying two different kinds of knowledge:

(1) the knowledge about hardware and software components, collected in the ASL Catalogue, and used at the basic level of the Language;

(2) the knowledge about combinations of hardware and software products into architectural units of various complexity (i.e. Units, Subsystems, Networks, and Architectures).

The complexity of both these kinds of knowledge justifies the use of a knowledge representation language for implementing the ASL. Moreover, the naturalness of semantic network formalisms in representing knowledge about a domain of strongly interrelated objects, motivates the use of a formalism of such kind in coping with the ASL Catalogue. The second kind of knowledge that must be handled by ASL, might have found in a rule based language a more appropriate representation scheme, as the R1 experience shows [McDe80]. However, the tight interaction of these two kinds of knowledge in ASL has suggested us the use of Procedural Semantic Network (PSN, see next Section). In doing so, we have retained the advantages of semantic networks, while encoding into classes and programs the rules for combining hardware and software components. A discussion on the logical adequacy of rules versus that of procedural semantic nets in modelling office information system architectures is clearly beyond the scope of this paper. However, when in Section 5 we will illustrate the most significant implementation details, we will show how the use of programs not necessarily implies a defeat of knowledge representation; at least in those languages, like PSN, where declarative and procedural knowledge coexist to model complementary aspects of knowledge

In the next Sections we will give an overview of the ASL components identified so far, after having introduced the implementation framework of the Language.

3.1. The Implementation Framework

The basic concepts of ASL, introduced in the previous Section, are implemented by using Procedural Semantic Network (PSN) [Leve79], a knowledge representation language that formalizes traditional semantic network concepts within a procedural framework. PSN provides the mechanisms for representing and manipulating objects and binary relationships between them, according to the modelling principles of object oriented languages. These principles can be summarized as follows:

(1) there is a one-to-one correspondence between the objects in the reality being modelled (in our case office information system architectures) and the model objects, and between the real world (binary) relationships and the model relationships;

(2) three of these relationships are factored out and used as abstraction mechanisms that permit the organization of the knowledge in the model; they are:

(2.1) the instance-of relationship, corresponding to the classification abstraction mechanism, by which objects with common properties are gathered into classes; an object is then an "instance-of" the class where it belongs; in turn, classes may be instances of metaclasses. Metaclasses, beside helping the organization of knowledge at a meta level, allow the definition of properties of classes, a feature that turns out to be very useful, as it will be shown in Section 5. Notice that this notion of class instance is fundamentally different from the notion of prototype instance, mentioned above;

(2.2) the part-of relationship, corresponding to the aggregation abstraction mechanism, by which an object is seen as the aggregate of the objects which are related to it; these relations can be further divided into structural (the ones that "constitute" the object, also called properties), and assertional (those that merely make an assertion about the object, and can be later retracted) [Wood75];

(2.3) the is-a relationship, corresponding to the specialization abstraction mechanism, by which a class of object is seen as a special case (or subclass) of another class; the former class is then "is-a" the latter.

The three abstraction mechanisms interact with each other by means of inheritance: a subclass inherits all the properties defined by its superclass, whereas an instance of a subclass is always an instance of its superclass.

Four operations are possible on PSN classes:

(1) create an instance of a class, realized by the to-put procedure associated to the class;

(2) remove an instance from a class, performed by the class to-rem procedure;

(3) get all the instances of a class, for which the to-get procedure is defined;

(4) test whether an object is an instance of a class, corresponding to the to-test procedure.

These procedures give the semantics of the class, as they interpret the class structure in the intended way. The PSN interpreter provides a standard procedure for each of the four operations, to be used when the semantics of a class is standard. However, a class may be given a non standard semantics by specifying 'ad hoc' programs to perform one or more of its four operations. We will see in Section 5 how this feature of PSN can be used in representing some kinds of knowledge involved in architecture modelling.

3.2. The ASL Catalogue

The ASL Catalogue is a knowledge base of objects representing the commercial hardware and software components which are typically employed in office information systems architectures. Following the representation paradigm of PSN, illustrated in the previous Section, the Catalogue includes one object for each component. Classes of components are defined on the basis of a functionality criterion, i.e. hardware and software products having the same functionalities are arranged within the same class. For instance, objects representing computers are classified in the Catalogue as instances of the same (hardware) component class, namely the class "Computer". The same applies to software components, with the class, say, "CentralizedDBMS", having as instances those objects representing centralized Database Management Systems.

Classes are taxonomically organized by mean of the is-a relationship, which defines a lattice, called the is-a hierarchy. The most general class of the Catalogue is the class "Catalogue", having as instances, by inheritance, all the Catalogue objects. The criterion which has been used for defining specializations of class "Catalogue", is again that of object functionality. Thus, for instance, class "Component", a specialization of "Catalogue" collecting all component objects, is specialized in "HardwareComponent" and "SoftwareComponent". In this sense, the is-a relationship is used in the Catalogue to (partially) order classes of components according to their functionality, with the most general class having the least specified functionality. The specialization of functionality reaches a level of granularity that is meaningful with respect to the aims of Architecture Design within the TODOS methodology. In other words, a class of Catalogue objects is a leaf of the is-a hierarchy when a further specialization of it would have not added relevant details from the TODOS methodology viewpoint. The Catalogue is-a hierarchy will be introduced in Section 5.2. Appendix A.2 shows the whole Catalogue is-a hierarchy.

Relationships between components are described in the Catalogue in two different ways: as properties (or slots) of the corresponding objects, if such relationships represent structural attributes of components (like the CPU model of a computer, or the external interfaces of a peripheral); as assertions on the corresponding objects if they represent time varying statements (like the price of a component).

The scenario that we have envisaged in designing the ASL, is one where the office system designer uses interactively the ASL interpreter to incrementally make up an Architecture, in a bottom-up fashion. As the objects defined in the ASL Catalogue are the basic ingredients of Architectures, the Catalogue must be made accessible through a powerful query language to serve its purpose within this scenario. Such query language must enable the office system designer to retrieve information about components through a number of different channels. In the next Section, we will illustrate in detail the features of the query language to operate on the Catalogue. What is relevant here, is that the user of ASL will see the Catalogue as a repository of information which he consults when creating architectural units.

In order to function, the Catalogue must be set up and kept updated, with new products or new versions of already existing products. To perform this task, we have defined an 'ad hoc' role, the Catalogue Administrator, who has the responsibility of initializing the ASL Catalogue and of maintaining it up-to-date. To carry out this task, the Catalogue Administrator must have an in-depth knowledge of the Catalogue structure, i.e. of the classes constituting the Catalogue and of their organization. The ASL System provides the Catalogue Administrator with a set of special operations to perform the Catalogue maintenance. These operations, to be described in more detail in Section 4, are accessible from the Administrator Interface, an interface based on the same principles of the ASL Interface.

3.3. The Unit level

Units are the atoms of Architectures, in that they are the lowest level building blocks of Architectures. A Unit is an object which represents a (hardware or software) component effectively used in an office information system. As such, a Unit refers directly to the Catalogue object that it 'materializes' as a part of an Architecture. Thus a Unit can be either a hardware or a software Unit, depending on the Catalogue component that the Unit realizes. The most relevant Units, from the Architecture specification viewpoint, are those representing computers and peripherals, which are the basic constituents of Architectures. But conceptually the model makes no difference between a computer Unit and the Units which represent parts of it, as, for instance, the computer's external interfaces or display. All these are Units as well, as they 'materialize' Catalogue objects.

The specification of an Architecture is done in a bottom-up fashion, starting from the definition of the Units that constitute the Architecture, and proceeding up to the higher level objects, Subsystems, Networks, and Architectures. This way of defining Architectures is imposed by the implementation language that has been chosen for ASL. In fact, a top-down Architecture specification methodology would imply the specification of higher level objects in terms of lower level objects, to be later defined. However, PSN does not allow the creation of an object that is related to undefined objects, and this definitely prevents the use of a top-down methodology for Architecture specification. We note that this limitation is not restrictive as far as the TODOS methodology is concerned, and can be afforded without suffering any problem.

Units can be created, removed, and queried to find out which are the Units' property values. An important operation that can be performed on a computer Unit is the Unit expansion. Expanding a computer Unit means (semantically) to increase one of the computer's functionalities by adding an appropriate device. This device can be a hardware or a software device. In the former case, we have a hardware expansion, whereas in the latter we have a software expansion. A typical hardware expansion is the addition of a memory board to a computer, to increase the computer's storage capacity. A software expansion is the installation on a computer of a software package, which enables the computer to perform one more function, or to perform better one of the computer's functions.

The notion of expansion captures an operation that the current technology has made very common in the practice of computer systems configuration. For this reason, a considerable numbers of such extensions are currently possible. An available expansion slot of a computer may be used to install an expansion board whose functionality may be as variable as the function performed by a program. Furthermore, a large variety of hardware components can be expanded, not only computers. For practical reasons, we have limited the hardware expansions modelled by the ASL to computer expansions of the following kinds:

(1) coprocessor, that is the installation on a computer of an additional CPU, usually done for increasing the computer's performance on calculations of a specific kind (numerical, graphical, and so on);

(2) main memory;

(3) diskette: most personal computers have a diskette expansion slot available in their base models; this kind of expansions obviously applies only to personal computers;

(4) fixed disk;

(5) cartridge: as in the case of diskette, this expansion only applies to computers that may have a cartridge driver;

(6) external interface: external interfaces are the adapters by which a computer can be point-to-point connected to another device; an expansion of disk kinds therefore enables a computer to augment its capabilities of connection;

(7) network interface: as for external interface, a network interface expansion results in the installation on an available computer expansion slot of an adapter which enables the connection of the computer in a computer network.

Of course, the conceptual machinery that has been employed to model these expansions, may analogously be used to extend ASL, so to include other hardware expansions, or expansions to peripheral Units. Software expansions are of only one kind, as it has been decided that it would not be meaningful to further categorize them.

Expansions are treated as assertional properties of Units, so that their value can be modified. Thus, a computer Unit has no expansions when it is created, but may have free expansion slots. If yes, the Unit can be later expanded by an appropriate expansion operation. As a result of an expansion, the Unit's expansion relation will have a new value.

An expansion operation involves a number of checks: the computer Unit being expanded must have a free expansion slot; the free expansion slot must be of the appropriate kind; most importantly, there must be an expansion board of the desired kind which is compatible with the Unit.

To perform the last checking in a way which is uniform with the model, expansion boards have been included in the ASL Catalogue, as hardware components of a special kind; in addition, special classes, generally called Compatibility classes, have been defined to represent knowledge about the compatibility of hardware and software components. There are two basic kinds of Compatibility classes. One of these kinds, consisting of Expansion Compatibility classes, deals specifically with expansions. The other kind is given by point-to-point Compatibility classes, and will be introduced in Section 3.4. There is one Expansion Compatibility class for each kind of expansion; for any hardware expansion, an instance of the associated Compatibility class tells which hardware component is compatible with which expansion board; for software expansions, the Compatibility class provides also additional information on the software required for making the expansion. The use of Compatibility classes in expansion operations should now be obvious: instances of the appropriate Compatibility class are looked up to find out whether a certain Unit (realizing a specific hardware component) is compatible with a certain expansion board.

An expansion previously made to a Unit, may be later retracted by using the Remove Expansion operation, also provided at the Unit level.

3.4. The Subsystem level

A Subsystem is an ASL object consisting of a set of point-to-point interconnected computer or peripheral Units, with the constraint that there be at least one computer Unit. Subsystems represent the simplest form of aggregation of components that can be found in an Architecture. Aggregates of increasing complexity are modelled in ASL at higher levels of the Architecture decomposition, namely the Network and Architecture levels. The presence of at least one computer in a Subsystem guarantees the 'autonomy' of the Subsystem. In fact, whether the Subsystem will be a host of a Local Area Network, or it will 'stand alone', thus representing the simplest form of Architecture, it must necessarily contain a computer in order to be able to operate.

Another important constraint on the definition of a Subsystem is that there be no isolated Units within the Subsystem. This constraint can be expressed more formally by viewing a Subsystem as an undirected graph whose nodes represent the Units and whose arcs represents the point-to-point connections of the Subsystem, and imposing the condition of seriality on the graph, which says that any node of the graph must be reachable from any other node.

In the real world, Subsystem connections are established through external interfaces. The ASL reflects this fact in a straightforward way: each Unit has a relation whose value is the set of the Unit's available external interfaces. When the Unit must be point-to-point connected to another Unit to form a Subsystem (or a part of it), the set of available external interfaces of both Units is checked; if a match is found, the point-to-point connection is established.

The creation of a Subsystem thus requires, among others, the following constraint checks:

(1) the check for seriality, which is a constraint on the topology of the Subsystem;

(2) the check for compatibility, which is a constraint on the feasibility of the point-to-point connections of the Subsystem.

The first check is performed by a simple algorithm, whose complexity is of the order of the square of the number of Units that constitute the Subsystem. The second check is done upon establishing each point-to-point connection of the Subsystem.

Point-to-point connections are ASL objects themselves. The creation of a point-to-point connection is always a side effect of the creation of a Subsystem, and is transparent to the user, who only specifies which Units are to be connected to make the Subsystem up. Point-to-point connections may be established between a computer and any kind of peripherals; in addition, magnetic disks may be connected between them. The compatibility of two Units is checked in a way that depends on the nature of the Units. When one of the two Units is a printer or a terminal, the compatibility is checked by matching the available external interfaces. This is due to the fact that this kind of connections is very common and depends exclusively on the involved interfaces. For the other kinds of connection, point-to-point (or PTP) Compatibility Classes, have been defined. Their use is identical to that of Expansion Compatibility Classes.

Other operations that can be performed on Subsystems are: the addition (removal) of a Unit to (from) an existing Subsystem; the addition (removal) of a connection to (from) an existing Subsystem.

3.5. The Network level

Subsystems may communicate in a multipoint fashion via Computer Networks. This aspect of Architecture Modelling is considered in ASL at the Network level.

The application domain of ASL, i.e. Office Information Systems, allows us to restrict to Local Networks, which typically provide interconnection of a variety of data communicating devices within a small area. Furthermore, among the three categories of Local Networks presented in [Stal84], the ASL includes only Local Area Networks (LANs), which are the most appropriate for the application domain being considered. To avoid any confusion between the LANs described in the ASL Catalogue, representing hardware machinery, and the LANs described at the Network level, representing aggregates of Subsystems, we will reserve the term 'LAN' for the former, and call the latter Office Networks.

Office Networks can be characterized in terms of topology, which can be a bus, ring, or tree topology. Accordingly, ASL objects representing Office Networks may be defined to be of one of these three kinds. What is common to the different Office Networks definitions is the property whose value gives the set of Subsystems that make the Office Network up. The other parameters of the definition depend on the topology of the Office Network being defined.

The creation of an Office Network object, requires the use of a Catalogue object, representing the hardware support (i.e. the LAN) on which the Office Network is based. In addition, each Subsystem that is to be included in the Office Network must have an 'escape Unit', that is a Unit which directly connects to the LAN. A Unit can be connected to a LAN if it fulfills the necessary hardware and software requirements. Thus, the creation of an Office Network object implies a compatibility check for each Subsystem that participates in the Office Network. This check is performed in a very simple way, just verifying that the network interface of the escape Unit of each Subsystem 'agrees' with the LAN that supports the Office Network. The same applies to the network software.

Other operations provided by ASL for Office Networks, beside the creation and removal operations, are the addition (removal) of a Subsystem to (from) an Office Network.

3.6. The Architecture level

ASL objects representing Architectures are at the highest level of the Architecture decomposition, i.e. the Architecture level. Architectures may be of two kinds: simple Architectures, which consist of just one Subsystem, and complex Architectures, which consist of a (non empty) set of interconnected Office Networks.

A simple Architecture can only be created and removed. Any other operation on the Subsystem that constitutes the Architecture must be performed before creating the Architecture, via the operations provided at the Subsystem level.

Complex Architectures must be explicitly created, even those consisting of one Office Network. The different Office Networks that constitute a complex Architecture are connected through Gates. A Gate is a Subsystem that has been declared to belong to more than one Office Network. As such, it is considered to link the Office Networks which it is a member of. The concept of Gate is not explicit in the ASL, as it is unnecessary, strictly speaking. Thus, any Subsystem which belongs to at least two Office Networks is potentially a Gate; it plays effectively the role of a Gate when all the Office Networks where it belongs are declared member of one Architecture.

After it has been created, an Architecture may be manipulated by adding to or removing from it one Office Network.

4. ASL Interface Operations

As already pointed out, the ASL System provides two main facilities:

(1) a knowledge base of commercial hardware/software components (Catalogue);

(2) an environment tool for specifying Architectures.

An interface is associated to each of them: an Administrator Interface for Catalogue maintenance (CA Interface) and an Office System Designer Interface (ASL Interface) to query the Catalogue and incrementally define an Office Architecture. In order to limit the complexity of the design phase, the interface provides a more abstract presentation of the knowledge base, ruling out the details that can be automatically handled. The available operations of the two interfaces will be briefly described in the next Sections. Appendix B contains a more detailed description of them.

4.1. Catalogue Administrator Interface Operations

The CA Interface provides operations to query the structure and the contents of the Catalogue and to modify it according to the hardware and software products availability.

The is-a hierarchy is one of the organizational structure of Catalogue contents. The following operation is provided to query this structure.

is-a

is-a returns the is-a hierarchy of the Catalogue classes.

According to the organizational object-oriented paradigm that has been used to organize the Catalogue knowledge base, each component is an instance of a class and it is described as the aggregation of other objects, its constituent parts. The following operations take these principles into account.

get_component Class Condition

get returns the instances of Class that satisfy Condition.

An empty condition is always true. The current implementation allows only conditions with the equality and set-membership predicates stated on the property values of objects. A two-level nesting is supported (see the query showed on figure 13.d).

display_class_definition Class

display_component Component

display_class_definition returns the Class properties, their domains and default values, if any. display_component displays the Component properties and their values.

instance_of Component

part_of Component

The instance_of operation is provided to get the classes which the given component is an instance of. The operation part_of returns for each Catalogue object x that has Component as value of some property or relation p, the pair <x, p>.

The Catalogue Administrator is allowed to add and remove components from the Catalogue in order to reflect the availability of commercial products.

create_component Class CompName AttributeValueList

delete_component Component

create_component creates a new Catalogue object whose name is CompName. The AttributeValueList is a list of <Property, Value> pairs that specify the value of each component property. delete_component deletes a component from the Catalogue. The aggregational structuring principle forces not to allow the deletion of an object that is part of an existing component. So if a delete_component is invoked on a component that is part of another object, the operation fails without any effect.

A 'version' of a component C is a component C' that differs from C at most for the value of any property different from "Vendor", "CompName", and "CompVersion". The concept of version has been introduced in ASL to model the cases where a property of an object may have several alternative values. In order to facilitate the handling of components that are versions of an already existing component, the following operations are provided.

create_version Component CompName PropValueList

delete_versions Component

get_versions Component

create_version creates a component whose property values are equal to those of the input component, except for the properties in PropValueList that take the new specified value. delete_versions cancels all the objects that are versions of the input object, i.e. that have the same values for the properties "Vendor", "CompName", and "CompVersion" as the input object. The operation does not have any effects on Component. Finally, get_versions returns all the Catalogue object that are versions of the given Component.

The compatibility classes are defined to declaratively describe the compatibility among different Units. As explained in Section 3.3, there are two kinds of compatibility classes which describe: (1) the compatibility among a computer Unit and its possible expansions (hardware and software), and (2) the compatibility between the expandable Units in a PTP connection.

create_<expansion>_compatibility CompName PropValueList

create_<ptp-connection>_compatibility CompName PropValueList

delete_<expansion>_compatibility CompName

delete_<ptp-connection>_compatibility CompName

The above operations are provided, for each of possible compatibility specification (the complete list is given in Appendix B), to create and delete compatibility descriptions. PropValueList is a list of pairs <Property, Value> that describes the new compatibility object. The display and selection of compatibility specifications are accomplished by means of the display and get operations available on all the Catalogue objects.

4.2. Architecture Designer Interface Operations

In this Section, the operations available to the Architecture designer for specifying Architectures are presented. At the Catalogue level, only queries are allowed by this interface. At any of the four levels for structuring Architectures, also operations for creating, removing and updating ASL objects are provided.

4.2.1. Catalogue

A subset of the CA Interface operations are available also at the Catalogue level of the ASL interface. These are the operations that allow the Architecture designer to access the Catalogue information but not to modify it. Thye are:

is-a

get_component Class Condition

display_class_definition Class

display_component Component

instance_of Component

part_of Component

get_versions Component

4.2.2. Unit level

The atomic Architecture components, computer and peripheral Units, are manipulated at this level. A Unit is created according to the Catalogue specifications. Operations are provided to expand computer and disk Units. The application of these operations fails if the proposed expansion is not allowed in the current Unit configuration, that is the Unit has no free expansion slots or none of the free ones is compatible with the chosen expansion. In such a way, the ASL System works as a designer assistant.

Operations similar to those at the Catalogue level are provided to query the objects of this level.

get_unit Class Condition

display_unit Unit

instance_of Unit

part_of Unit

The Unit level provides also facilities to create and manipulate the basic components of a Subsystem, that is computer and peripheral Units.

create_unit UnitName Component

delete_unit Component

The creation of a Unit causes the creation of all the Units that are part of it. For example, the creation of a new computer Unit implies the creation of the following: (1) a set of external and network interface Unit objects, each of which models a physical external interface of the computer; (2) a set of expansion slot Units; and (3) its built-in peripheral Units, that is the display, the keyboard, the pointing-device and the storage devices. All the computer and peripheral parts are dependent from the computer they belong to, that is it is not possible to manipulate them directly. Any modification of them is a side effect of the manipulation of their computer. However, it is allowed to select and display them.

A delete operation is provided to cancel Units. As side effect, all the Unit which are parts of the Unit being deleted are also deleted. A computer or a peripheral Unit cannot be deleted if is part of some existing Subsystem.

Initially, a Unit has all the functionalities of the corresponding Catalogue component. In addition, all its interface and expansion slot Units are free, i.e. they can be used for ptp-connections and expansions, respectively. The computer Unit functionalities can be increased by adding hardware and software expansions. For such reason a set of expansion operations have been provided, one for each possible expansion.

expand_with_main_memory_board ComputerUnit Amount

expand_with_coprocessor_board ComputerUnit Functionality expand_with_external_interface_board ComputerUnit ExtInterfaceType

expand_with_network_interface_board ComputerUnit NetInterfaceType

expand_with_fixed_disk_board ComputerUnit FixedDisk

expand_with_diskette_board ComputerUnit Diskette

expand_with_cartridge_board ComputerUnit Cartridge

expand_with_software ComputerUnit SoftwareTool

The above family of operations is provided to expand computer Units. A computer Unit allows hardware and software expansions. Each software product can be used as expansion, except operating systems. For each hardware expansions, parameters are required to allow the system to select automatically the appropriate expansion board. For a main_memory_board expansion, for example, the final amount of memory must be specified. The selection of the appropriate expansion board, if any, is automatically done by the ASL implementation.

For each of the expansions above listed, a remove operation is provided. For instance, the operation to remove a main memory board is:

remove_main_memory_board ComputerUnit Amount

The operations to remove the other expansions are similar, each of them requires the same input parameters of the corresponding expand. A complete list of them can be found in Appendix B.

4.2.3. Subsystem level

Subsystems are the simplest form of architectural Units aggregation and they are used to create higher level objects such as Networks and office Architectures. They are modelled as aggregates of serially interconnected Units, at least one of which is a computer Unit. As a particular case, a single computer Unit is a Subsystem.

The operations to get all the Subsystem objects that satisfy a particular condition and to display a Subsystem object are similar to those of the others levels.

get_subsystem Subsystem Condition

display_subsystem PTPConnectedSubsystem

The language automatically treats each computer Unit as a Subsystem. To create complex Subsystems, that is sets of point-to-point interconnected Units, the following operations are available.

create_subsystem SubsName, Set of Units, PTPConnections

delete_subsystem Subsystem

In order to create a new Subsystem, the create_subsystem operation must first create the PTP connection objects that are required to define the new Subsystem. As for the Units that are parts of a computer or a peripheral Unit, the PTP connection objects cannot be directly manipulated, they are created and deleted only as side effect of operations on the Subsystem they belong to.

A Subsystem must satisfy the following constraints: (1) the set of component Units must contain at least one computer Unit; (2) the Subsystem must be serially connected and (3) there must be at most one connection between the same pair of Units. If the specification of the Subsystem to be created does not satisfy these conditions the create_subsystem fails with no effect. create_subsystem also requires that the Units involved in each PTP connections are PTP-connectable, that is have free compatible external interfaces. The Units involved in a point-to-point connection are modified making used one of the free compatible interfaces. If such interfaces are not available, the operation fails. A further constraint requires that a Unit cannot be part of more than one Subsystem. The reason for this limitation follows from what has been written above. A Unit free interface is marked "used" each time it participates in a connection, so if a Unit is shared among different Subsystems each Subsystems is effected by this change. It might be, for example, that a connection cannot be established because an interface has been used for a connection in a different Subsystem.

delete_subsystem deletes a Subsystem. The Units that belong to the Subsystem are not deleted by this operation, but their external interface Units that participated in the Subsystem PTP-connections are made free.

As already point out, a Subsystem is modelled as a set of expandable Units and a set of point-to-point connections among them. Operations are provided to add or remove each of these Subsystem components.

add_unit_to_subsystem Subsystem, Unit, PTPConnections

add_ptp_connection_to_subsystem Subsystem, PTPConnection

remove_unit_from_subsystem Subsystem, Unit

remove_ptp_connection_from_subsystem Subsystem, PTPConnection

All the above operations modify a Subsystem through the addition and removal of a Unit or a PTP connection. The resulting Subsystem must obviously satisfy all the constrains listed above.

Sometimes it might be helpful to be able to check that some conditions are satisfied in order to be sure not to violate the Subsystem constrains. For such reason, the following predicates are provided.

in_subsystem Unit

ptp_connected_subsystem Subsystem

in_subsystem is a predicate that returns true if a Unit is a component of some existing PTP connected Subsystem. ptp_connected_subsystem returns true if the input Subsystem is PTP connected, false otherwise.

4.2.4. Network level

A Network is modelled as a set of Subsystems interconnected through a local area network. The Network level provides operations to create, delete and manipulate bus, ring and tree Office Networks. The get and display operations are also provided at this level.

get_network Condition

display_network Network

A Network is specified as the aggregation of a communication device and of a set of Subsystems, each of which is connected to the communication device by means of one of its component Units. Operations are provided to create and delete each of the three kinds of Networks.

create_bus_network NetName, LAN,
Set of (Subsystem, ConnectionUnit),
List of Subsystems

create_ring_network NetName, LAN,
Set of (Subsystem, ConnectionUnit),
List of Subsystems

create_tree_network NetName, LAN,
Set of (Subsystem, ConnectionUnit),
List of Subsystems

delete_bus_network BusNetwork

delete_ring_network RingNetwork

delete_tree_network TreeNetwork

Three network creation operations are provided to create Networks with a "bus", "ring" or "tree" topology. The input LAN parameter specifies the basic model of the used Network communication device, while the list of Subsystems specifies the order in which the Subsystems are placed along the communication device. The pairs <Subsystem, ConnectionUnit> describe for each Subsystem which is the Unit used to physically connect the Subsystem to the Network.

A Subsystem can be a Network host only if the given connection Unit has a free network interface Unit that turns out to be compatible with the chosen communication device. The creation of a new Network has the side effect of making used such selected interface Unit. The deletion of the Network makes it available again.

As for the others levels, operations are provided to add and remove the objects that are parts of the objects that are defined at this level, that is operations are available to add and remove Subsystems from Networks.

add_subsystem_to_bus_network Bus_Network, (Subsystem1, ConnectionUnit), Subsystem2

remove_subsystem_from_bus_network Bus_Network, Subsystem

Similar operations are provided for the other two kinds of Network. The add operation modifies the Network by adding Subsystem1 immediately next to Subsystem2. ConnectionUnit is required in order to specify the connection point between the Network and Subsystem1.

The remove operation removes Subsystem from the Network. It fails if Subsystem is the only Subsystem of the Network.

4.2.5. Architecture level

An Architecture is defined to be either a single Subsystem or a not empty set of fully interconnected Networks that share common Subsystems. The common Subsystems have been referred as "gates".

The Architecture level makes available operations to query, create, delete and modify the objects of this level.

get_architecture ComplexArchitecture

display_architecture ComplexArchitecture

The system automatically treats a Subsystem as a simple Architecture. To create and delete complex Architecture the following operations are available.

create_architecture ArchitectureName, set of Network

delete_architecture ComplexArchitecture

A create operation fails if one of the constituent Networks already belongs to an office Architecture (a Network cannot be shared among different Architectures).

Operations are also provided to modify the Architecture components.

add_network_to_architecture Architecture, Network

remove_network_from_architecture ComplexArchitecture, Network

The remove operation fails, without any effect, if the input Network is the only one of the specified Network.

Finally, a predicate to check if an Architecture is complex is provided.

complex_architecture Architecture

complex_architecture returns true if Architecture contains at least one Network, nil if it is a single Subsystem.

5. An ASL Implementation

In this Section we will present the main features of the PSN implementation of ASL, and describe how the operations illustrated in the previous Section have been realized on top of PSN. The reason why the description of the ASL operations has been separated from the description of their implementation should be obvious. Any programming language, other than PSN, might clearly be used for the implementation. However, as we have already argued, the expressive power of a knowledge representation language seems extremely appropriate to deal successfully with the domain under consideration.

As a notation convention, we will enclose between double quotes names of PSN entities, whether metaclass, class, object or property names. The names that we will quote in this Section may not be the real names used in the knowledge base, and this is due to the fact that, in order to avoid collisions, real names are sometimes awkward and counterintuitive.

5.1. Meta Level Definitions

PSN allows the definition of metaclasses, i.e. classes having classes as instances. Metaclasses have been defined in ASL for two reasons. First, they can be used as 'handles' for set of classes; this turns out to be very useful in dealing with higher order functions, as the enforcement of constraints like 'there must exist a compatibility class such that ...'; furthermore, having a handle for all ASL classes, permits the separation between the ASL conceptual machinery and the rest of the PSN language. The second reason is that metaclasses allow the definition of properties of classes, that is properties that apply not to single objects but to collections of objects as a whole (an example of this are the properties defined for compatibility classes, explained later).

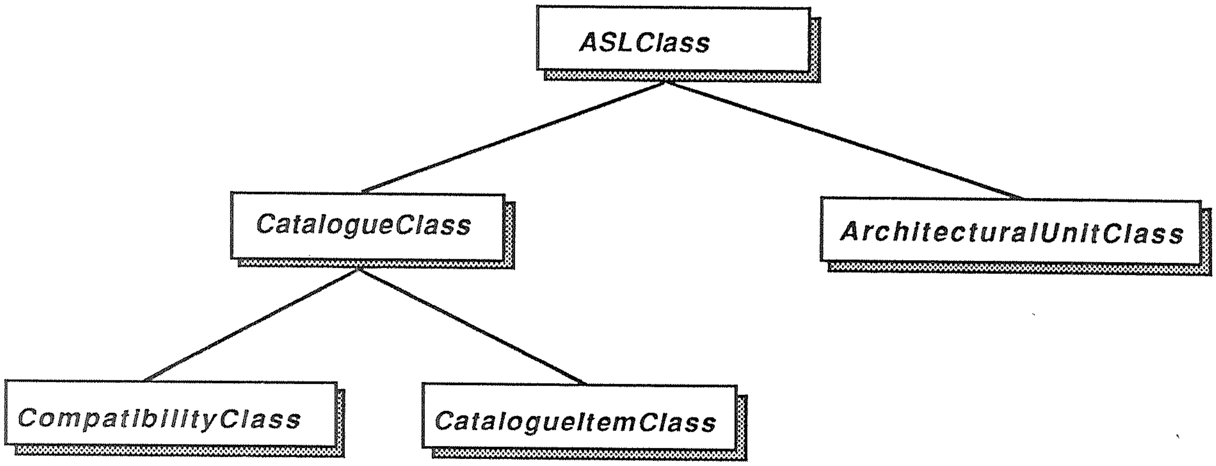
The most general ASL metaclass is "ASLClass", having as instances all ASL classes. In order to realize the ASL modularization, "ASLClass" is specialized into "CatalogueClass" and "ArchitecturalUnitClass", the former being the metaclass of the classes constituting the ASL Catalogue, while the latter has as instances the classes corresponding to the four levels of Architecture abstraction (Unit, Subsystem, Network, and Architecture). In turn, "CatalogueClass" has two specializations: "CompatibilityClass", the metaclass of all compatibility classes, and "CatalogueItemClass" the metaclass of classes modelling Catalogue components. The is-a hierarchy at the metaclass level is shown in figure 1.

Compatibility classes represent the knowledge about the compatibility between components, which is used in making expansions to computers or in establishing point-to-point connections between hardware devices. The procedures that perform these tasks need to know which compatibility class represents compatibility information about which classes of components. This kind of information must be associted to whole classes of compatibility, and, to this end, the meta-class "CompatibilityClass" defines two properties. The value of these properties gives, for each compatibility class, the type of the devices whose compatibility is described by the class instances.

5.2. The Catalogue Items Knowledge Base

The most general Catalogue item class is "Catalogue", an instance of "CatalogueItemClass", which has as instances all the Catalogue objects. "Catalogue" is specialized into "Component" and "AuxCatalogue", where the former is the most general component class, while the latter has as instances all the objects that are necessary for the definition of Catalogue objects but do not represent components in the proper sense. An example of auxiliary objects are those representing vendors of hardware and software products, power requirements of components, or data models of database management systems. These objects are not of particular interest for the description of the ASL implementation, therefore we will not enter into the details of class "Aux-Catalogue".

The class "Component" defines three properties, "Vendor", "CompName", and "CompVersion", with the obvious meaning. These properties are inherited by all classes describing components, as these classes are specializations of "Component". The relation "Price" is defined to have "Component" as domain, and numbers as range. "Component" is specialized into "HardwareComponent" and "SoftwareComponent", thus reflecting a natural categorization of components.



ASLClass
CatalogueClass
ArchitecturalUnitClass
CompatibilityClass
CatalogueItemClass

Figure 1

5.2.1. Software Components

Figure 2 shows the offspring of class "SoftwareComponent" in the class is-a hierarchy. The first subdivision of software components is between operating systems (instances of "OperatingSystem"), and generic software tools (instances of "SoftwareTool"). In turn, the class "SoftwareTool" is specialized into "SystemSoftware", the class describing basic software packages, like programming languages compilers and interpreters, editors, network software and so on; and "OfficeSupportTool", whose instances are the packages that support office activities. Such packages are categorized on the basis of the processed information type, hence there are, as specializations of "OfficeSupportTool", the classes "VoiceProcessingTool", "ImageProcessingTool", and "DataProcessingTool". Database management systems, word processors, spreadsheets, calendar and scheduling packages are included in the offspring of "DataProcessingTool", while the only specialization of "ImageProcessingTool" is "GraphicTool".

This view of the world of software components must be ascribed to the aim of Architecture Modelling within the TODOS methodology, which focuses on the use of software packages for implementing an office information system. In this context, software packages with functionalities not pertaining to office automation are clearly of no interest, thus they have been excluded from the modelization.

5.2.2. Hardware Components

Hardware components have been subdivided into expandable components and static components. Expandable components include computers and peripherals, even though, for reasons mentioned earlier, the ASL model only allows for expansions to computers. Static components are those hardware devices which play a somewhat secondary role in the definition of Architectures; with the exception of LANs, they basically represent parts of expandable components. Static components are: local area networks, expansion slots, expansion boards, interfaces, and CPUs. Each of these is represented by a class, which is a direct specialization of class "StaticComponent", whereas classes "Computer" and "Peripheral" represent expandable components, and are specializations of "ExpandableComponent". Both "StaticComponent" and "ExpandableComponent" are "is-a" "HardwareComponent".

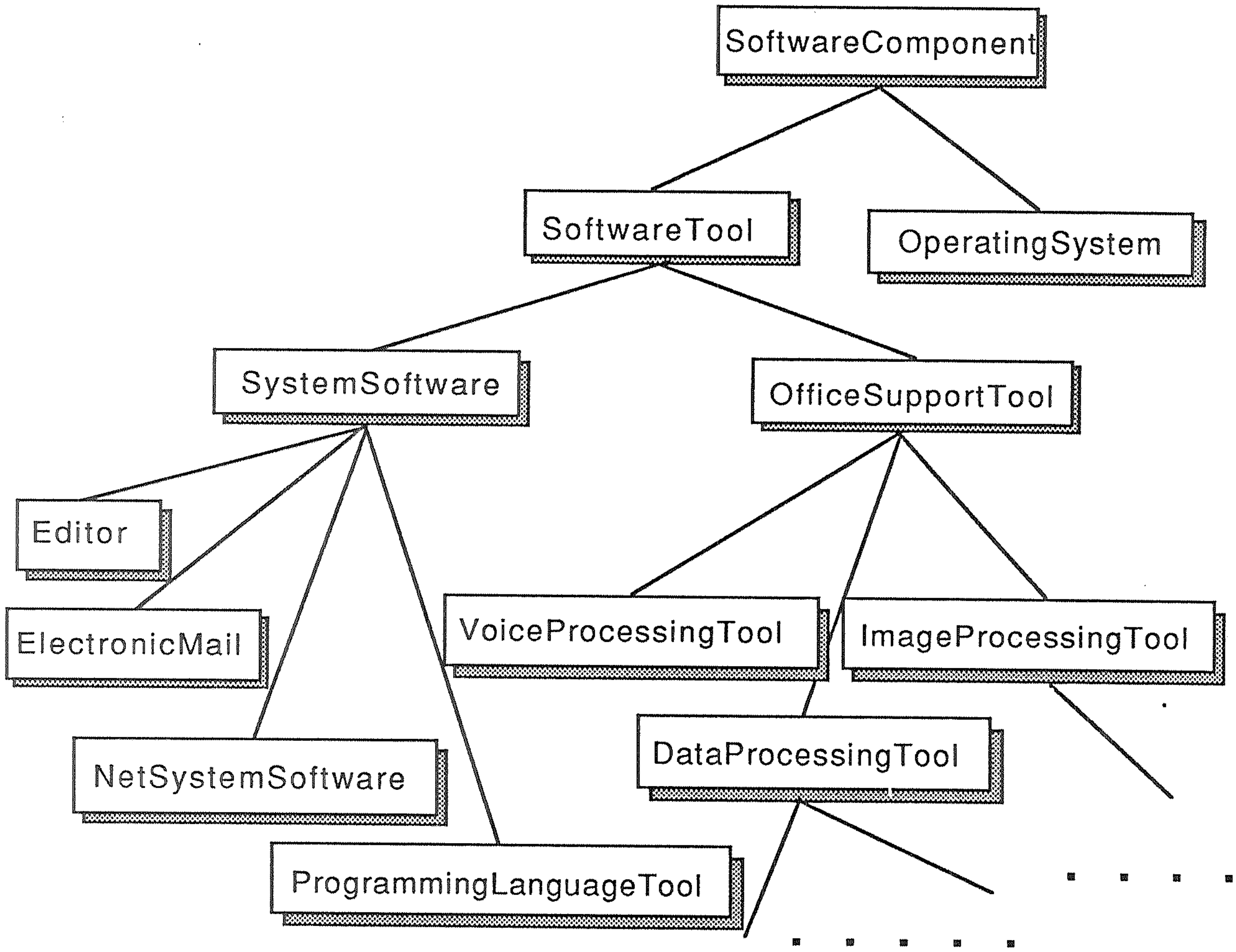

SoftwareComponent
SoftwareTool
OperatingSystem
SystemSoftware
OfficeSupportTool
Editor
ElectronicMail
VoiceProcessingTool
ImageProcessingTool
NetSystemSoftware
DataProcessingTool
ProgrammingLanguageTool

Figure 2

5.2.2.1. Local Area Networks

Local area networks (LANs) are modelled by ASL as instances of class "LocalAreaNet"; they are not to be confused with Office Networks, which represent the Architectural Units defined at the Network level, illustrated in Section 3.5. As explained earlier, there are three kinds of LANs in the ASL Catalogue, distinguished by topology: bus, tree and ring. Each kind of network is represented by a class, which is "is-a" "LocalAreaNet".

The properties defined for LANs may be divided into two sets. The first set consists of properties concerning the Network's attributes, both at the physical level (like the primary medium, or the transmission rate), and at the logical level (like the access protocol, or the software running the network). The second set of attributes represent topological constraints that must be satisfied when the Network is used in an Architecture; examples of these properties are the maximum and minimum distance between adjacent nodes, and the number of addressable nodes.

5.2.2.2. Interfaces

Interface objects are instances of class "Interface", and represent the adapters used for point-to-point connecting hardware devices. For instance, the connection between a computer and a disk is modelled in ASL (as it will be described later) by creating an object that relates the interfaces of the computer and of the disk being physically employed in establishing the connection.

Interfaces may be of two kinds: external or network interfaces. The former are instances of class "ExtInterface", and represent generic adapters for local connections. The latter are instances of "NetInterface", and model adapters for connecting a device to a local area network. Both "ExtInterface" and "NetInterface" are specializations of class "Interface".

The only additional property defined for interfaces, besides those inherited from "Component", is one that gives the type of the interface. This property ranges on the class of interface types, which deserves some explanation. The relation between interfaces and interface types is many to one, in the sense that many interfaces may be of the same type, but the type of an interface may be only one. For instance, a terminal server has many interfaces, all of the same type, i.e. RS232. Interface types are instances of class "InterfaceType", an auxiliary class, which is specialized by network and external interface types. The latter is further divided into three categories: bus (like SCSI ports), parallel (like CENTRONIX), and serial (RS232, for instance) interface types. A property common to all these types is one which tells whether the interface type is input, output or both. Parallel and Serial types have other properties describing physical features of the interface, like the minimum and maximum rate. The use of the properties of interface types in creating point-to-point connections is described later.

5.2.2.3. Expansion Slots

Expansion slots (instances of class "ExpansionSlot") represent slots of hardware devices, i.e. locations available on devices for the installation of expansion boards. Expansion slots may be of several kinds, depending on the type of board that can be placed in the slot. Each kind is modelled by a class which is a specialization of "ExpansionSlot". The ASL Catalogue includes the following kinds of slots:

(1) network interface, the slots in which a network interface board can be installed, in order to allow the hardware device to be connected in a computer network;

(2) external interface, which is the kind of those slots where an adapter (or external interface) can be installed; adapters are used for connecting hardware devices between them in a point-to-point fashion (the details on point-to-point connections are given in the Section on Subsystems);

(3) coprocessor, the slots used for expanding a computer with an additional coprocessor (expansions are explained in detail in the Section on Units);

(4) storage, the kind of slots available for storage expansions; these expansion slots are further categorized as cartridge, fixed disk and diskette expansion slots;

(5) generic, the kind of expansion slots that can be of more

than one of the four kinds above; in fact, it is very common that, among the expansion slots of a computer, few of them are 'ad hoc', i.e. dedicated to one kind of expansion, whereas the most allow for several kinds of expansion. An object representing a slot of the generic kind is made instance of the class of generic expansion slot; in addition, it is also an instance of the classes representing the kinds of slots it allows.

The categorization of expansion slots classes is summarized in figure 3.

5.2.2.4. Expansion Boards

Expansion boards are the hardware components that are installed in the appropriate expansion slots to realize expansions. For this reason, the organization of expansion boards classes strictly reflects that of expansion slots. Thus, we have class "ExpansionBoard" as the most general expansion board class, which is specialized by network interface, external interface, coprocessor, storage, and main memory boards classes. Storage expansions boards are further divided into cartridge, fixed disk, and diskette boards.

The reason why there are main memory expansion boards while there are no main memory expansion slots, is that the internal configuration of the main memory of a computer may be very complex (see the internal main memory configuration of the IBM PC RT, presented in Section 7.1.4, as an example). This is due to the fact that usually a computer has several main memory slots and several main memory boards that can be placed onto them. The representational problem arises because not all possible configurations (i.e. assignments of boards to slots) are allowed, but only a subset of them. Despite the fact that such subset is usually not very large, a complex conceptual machinery is necessary to represent faithfully this situation, including the complexity added to the programs that have to handle main memory expansions. Given the character of pre-competitive study of ESPRIT, and the Workplan constraints on WP4 within the TODOS Project, we have simplified the modelization of main memory expansions, treating them in a special way. We will describe in detail this treatment later in Section 5.4.2, here we note that our treatment does not cause any loss of generality or correctness to our model of office Architectures.

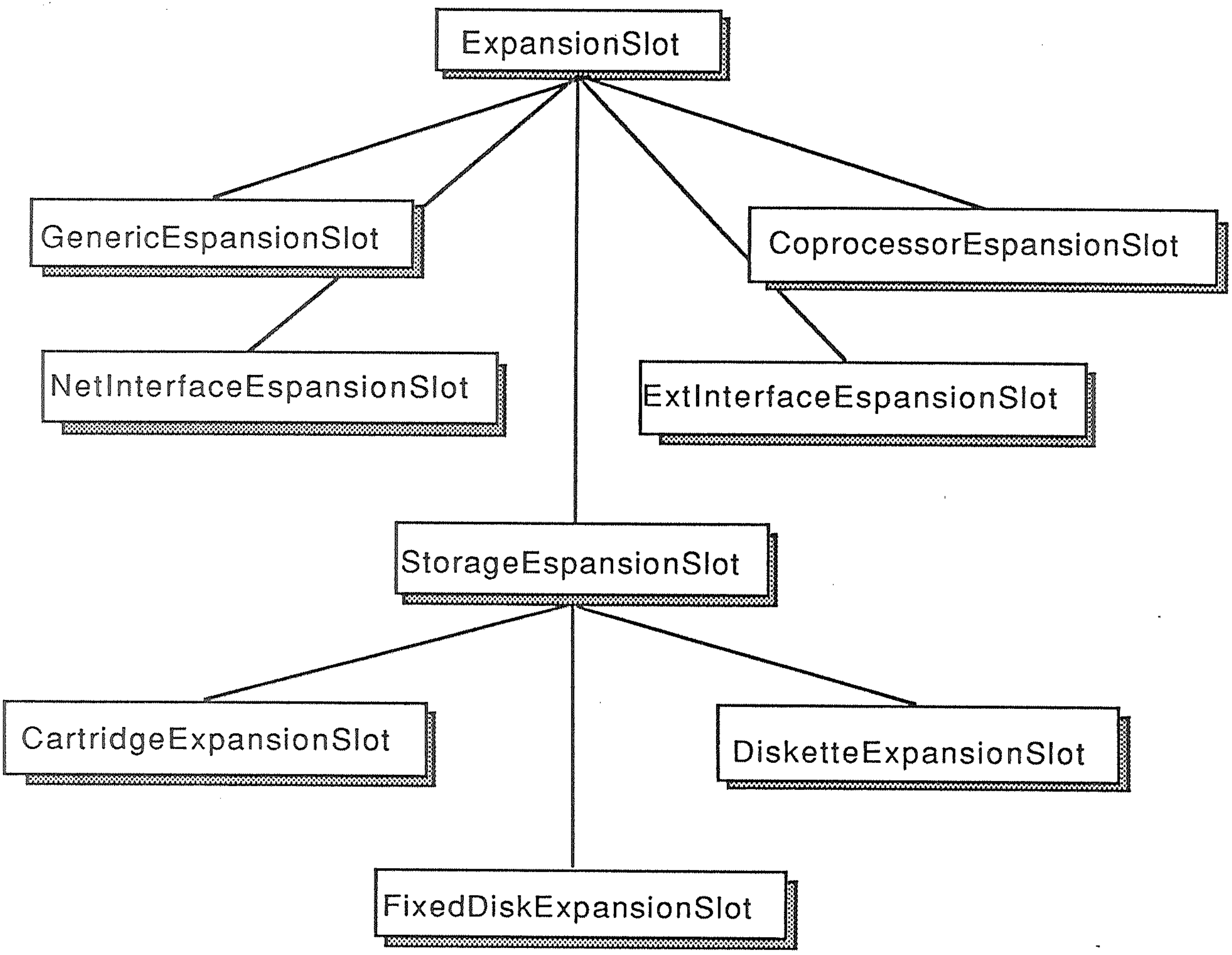

ExpansionSlot
GenericEspansionSlot
CoprocessorEspansionSlot
NetInterfaceEspansionSlot
ExtInterfaceEspansionSlot
StorageEspansionSlot
CartridgeExpansionSlot
DisketteExpansionSlot
FixedDiskExpansionSlot

Figure 3

As components, expansion boards inherit the properties defined by class "Component", described above. Relevant properties specifically defined for expansion boards are: for network and external interface board, there is one property whose value gives the set of interfaces carried by the board. Storage boards have a property which describes the storage device associated with the board. The quantity of main memory resulting from the installation of an expansion board is a property of main memory boards. Finally, coprocessor boards have two properties: one gives the CPU cabled in the board, the other the functionality of that CPU, i.e. the kind of computation specifically performed by the coprocessor (numerical, graphical, and so on).

5.2.2.5. Expandable Components

The class of expandable components includes the most important hardware components of the ASL Catalogue, that is computers and peripherals, instances of classes "Computer" and "Peripheral", respectively. Computers, in particular, are the 'kernel' of ASL, as they can be considered the basic ingredient of Architectures. A computer is modelled as an object, which describes the basic configuration of the computer, that is the minimal hardware and software equipment that is sold atomically. The most relevant properties of a computer object are:

- "CPU": the CPU object of the computer, which is explicitly modelled because it may be important in the performance evaluation phase;

- "BuiltInCoprocessors": the set of built-in coprocessors;

- "MainMemory": a number that gives, in kilobytes, the total main memory of the computer;

- "UserAvailableMainMemory": the quote of main memory which is available to the user;

- "MaxMainMemory": the maximum amount of main memory that the computer can support;

- "TotalRAStorage": the size, also this in kilobytes, of the random access secondary storage of the computer; if the computer has more than one built-in secondary storage devices, the value of this property is clearly the sum of the sizes of the storage of each device;

- "ExternalRAStoragePeripherals": the set of external random access storage peripherals of the computer; it is important to represent explicitly this kind of storage devices because any of them can be autonomously point-to-point connected to another device, thus determining an implicit point-to-point connection of the computer;

- "InternalRAStoragePeripherals": the set of random access storage peripherals which are internally installed on the computer; these peripherals cannot be connected to other devices, as they have no visible external interfaces;

- "InternalSAStoragePeripherals": the set of the computer's internal sequential access storage peripherals;

- "ExternalSAStoragePeripherals": the set of the computer's external sequential access storage peripherals;

- "ExpansionSlots": the set of expansion slots available for expanding the computer;

- "NetworkInterfaces": the set of the computer's network interfaces;

- "ExternalInterfaces": the set of the computer's external interfaces;

- "OperatingSystem": the object representing the operating system running the computer;

- "AvailableLanguages": the set of programming language tools (i.e. compilers and interpreters) that are provided with the basic configuration of the computer;

- "AvailableEditors": the set of editors, where each editor is characterized in terms of the user interaction mode;

- "AvailableNetSoftware": the set of software packages that enable the computer to participate in a local area network;

- "AvailableOfficeSupportTools": the set of office support tools.

Each computer commercially available is represented in the Catalogue by one instance of "Computer". This creates some practical problems, due to the fact that usually there is not one basic configuration of a computer, but rather a set of alternative basic configurations that differ one another for some options. For instance, as it is shown in Section 7.1, the basic configuration of the IBM PC RT may have one of five alternative displays, and, as internal fixed disk, a 40 or a 70 Mbytes disk. All these possible basic configurations must be represented in the Catalogue as distinct objects, and this may be heavy for the Catalogue Administrator. The concept of version, already presented in Section 4.1, has been defined in ASL to help in these situations.

The class of all computers, "Computer", is specialized according to the various kinds of computers that have been considered meaningful for the modelization of office information system Architectures. These are: main frames, minicomputers, personal computers, integrated workstations, word processors, and network servers.

Peripherals are instances of class "Peripheral". A peripheral can neither be directly connected to a LAN, nor it can be expanded. Moreover, peripherals do not have built-in processors, and no property concerning software is defined for peripherals. Thus, of the above properties defined for a computer, class "Peripheral" only has property "ExternalInterfaces". External peripheral may have an internal sequential access peripheral, a fact that is representing by defining an appropriate property ("InternalSASPs") for class "ExternalStoragePeripheralT".

The class "Peripheral" is subdivided into storage and input/output peripherals. Figure 4 presents the is-a offspring of class "StoragePeripheral", the most general storage peripheral class, which defines the "AccessType" and "FormattedStorageCapacity" properties, with the expected meaning. As figure 4 shows, storage peripherals are categorized as internal and external peripherals. Internal storage peripherals represent the storage devices that are internal to computers. This is the case, for instance, of fixed disks of personal computers or workstations. These peripherals differ from external peripherals in that they do not have available external interfaces for being connected to other devices. External peripherals may as well be sequential or random access, and make their external interfaces visible. Random access storage peripherals, whether internal or external, are categorized as diskette, optical disk, and magnetic disk drivers. Instead, sequential access storage peripherals are

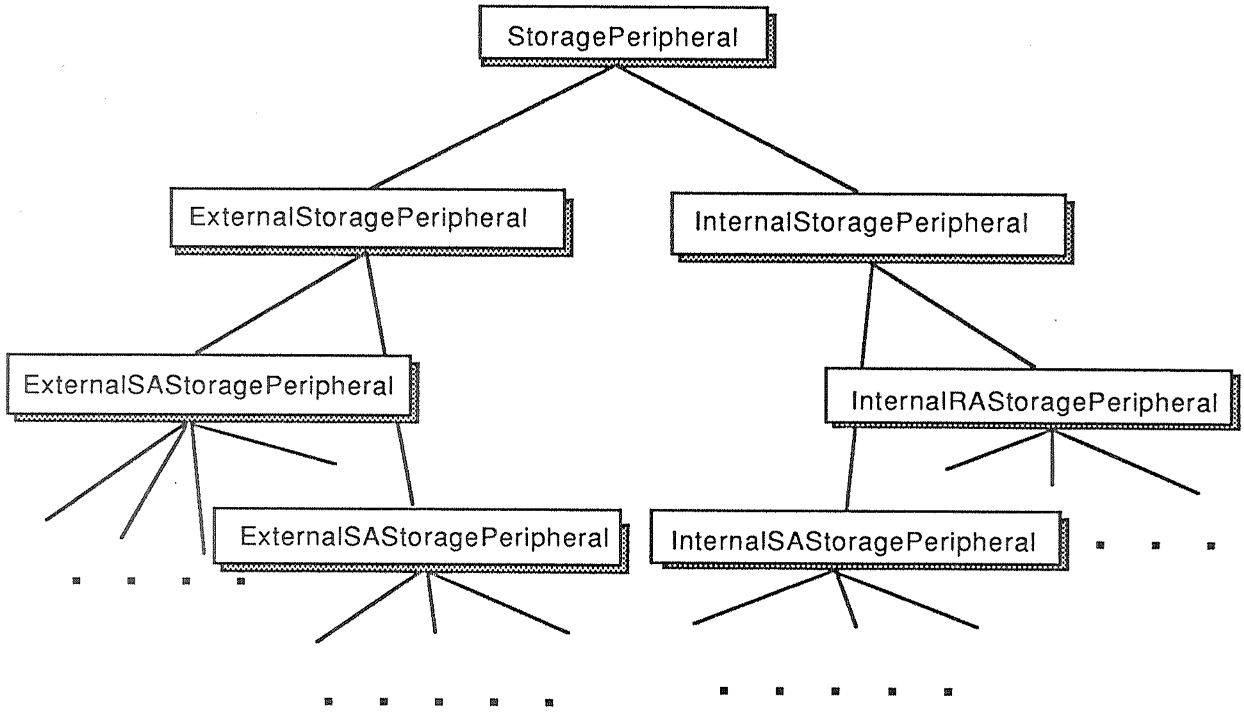

StoragePeripheral
ExternalStoragePeripheral
InternalStoragePeripheral
ExternalSAStoragePeripheral
InternalRAStoragePeripheral
ExternalSAStoragePeripheral
InternalSAStoragePeripheral

Figure 4

cassette, cartridge, and tape drivers.

The direct specializations of "IOPeripheral", the most general class of input/output peripherals, model the various kinds of such peripherals that are represented in the ASL Catalogue. These are: terminals (further divided into printer and video terminals), printers, plotters, scanners, displays, keyboards, and pointing devices.

5.3. Compatibility Classes

Compatibility classes can be considered members of the ASL Catalogue, as they describe a basic property of hardware and software components, namely the ability that components have of combining between them to generate 'complex' components. For this reason, the metaclass of all compatibility classes ("CompatibilityClass") is a specialization of "CatalogueClass". However, at the class level, compatibilities and catalogue objects are kept separated, to avoid any interference between products and their compatibility. This separation is realized by making the most general compatibility class ("Compatibility") an instance of metaclass "CompatibilityClass".

As already explained earlier, there are two kinds of compatibilities: expansion compatibilities, describing which expansion boards or software packages can be installed on which computers, and point-to-point compatibilities, which relate hardware devices that can be point-to-point connected. Accordingly, class "Compatibility" is specialized into "ExpansionCompatibility" and "PTPCompatibility", where the former is the most general expansion compatibility class, and the latter is the most general point-to-point compatibility class.

5.3.1. Expansion Compatibility Classes

The is-a offspring of "ExpansionCompatibility" is illustrated in figure 5. Expansion compatibilities are first categorized as hardware and software compatibilities, which are instances of classes "SoftwareExpansionCompatibility" and "HardwareExpansionCompatibility", respectively.

An instance of "SoftwareExpansionCompatibility"

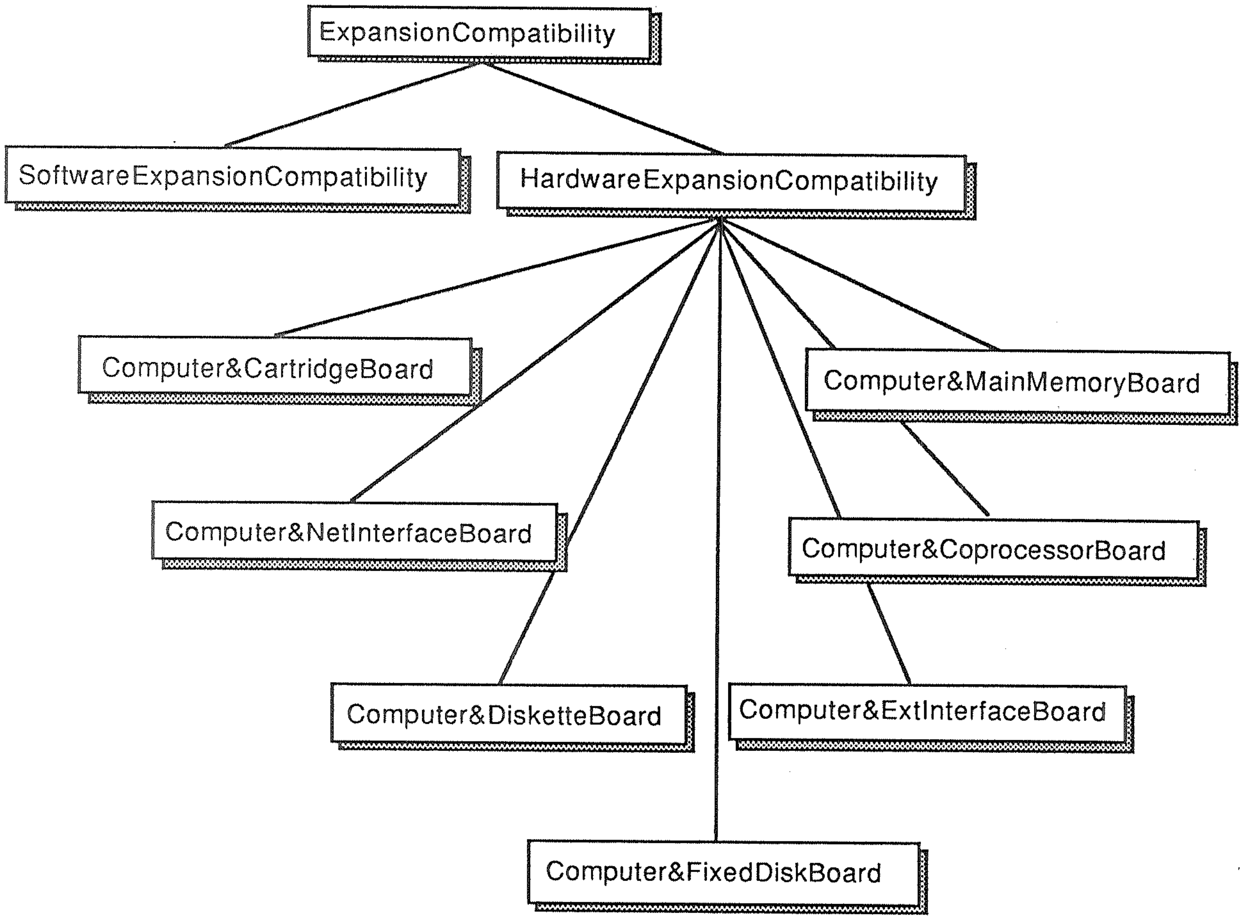

ExpansionCompatibility
SoftwareExpansionCompatibility
HardwareExpansionCompatibility
Computer&CartridgeBoard
Computer&MainMemoryBoard
Computer&NetInterfaceBoard
Computer&CoprocessorBoard
Computer&DisketteBoard
Computer&ExtInterfaceBoard
Computer&FixedDiskBoard

Figure 5

describes the fact that a certain software package (represented by the value of property "TargetSoftware" of the instance) can be installed on a certain computer (the value of "Machine"), provided that the computer has a set of required software packages (the value of "Required-Software").

There is one hardware expansion compatibility class for each hardware expansion that can be made to a computer. An instance of any such classes represents the fact that a certain computer (given by the value of property "Computer" of the instance) can be expanded with a certain expansion board ("Board"), which can be placed in one of a set of the computer's expansion slots ("Slots"), provided that the number of such boards already installed on the computer does not exceed a prefixed number ("MaximumNumber").

A non-standard to-test procedure is attached to each expansion compatibility class: instead of testing whether a certain instance belongs to a certain compatibility class (what the standard procedure would have done), the non-standard to-test receives as input a computer and a board, and returns 'true' if they happen to be compatible, and 'false' if not. Such procedure searches the extension (that is the set of instances) of the proper expansion compatibility class, to see whether there exists an instance that asserts the compatibility of the computer and the board being tested; it also checks the constraints implicitly represented by the "Slots" and "MaximumNumber" properties. This procedural attachment permits the representation of the semantics of each compatibility class within a 'black box' (i.e. the to-test procedure) whose service is accessible in a standard way. Thus, even though this kind of knowledge is represented procedurally, the application of abstract data types principles guarantees the encapsulation of such knowledge in its proper context.

5.3.2. Point-to-point Compatibility Classes

These classes are specializations of "PTPCompatibility", and are in a one-to-one correspondence with the pairs of point-to-point connectable devices. For practical reasons, we have restricted our attention to connections having either a computer at one end, or between two magnetic disks; the language can clearly be extended to treat all possible cases by using the conceptual modellization being illustrated.

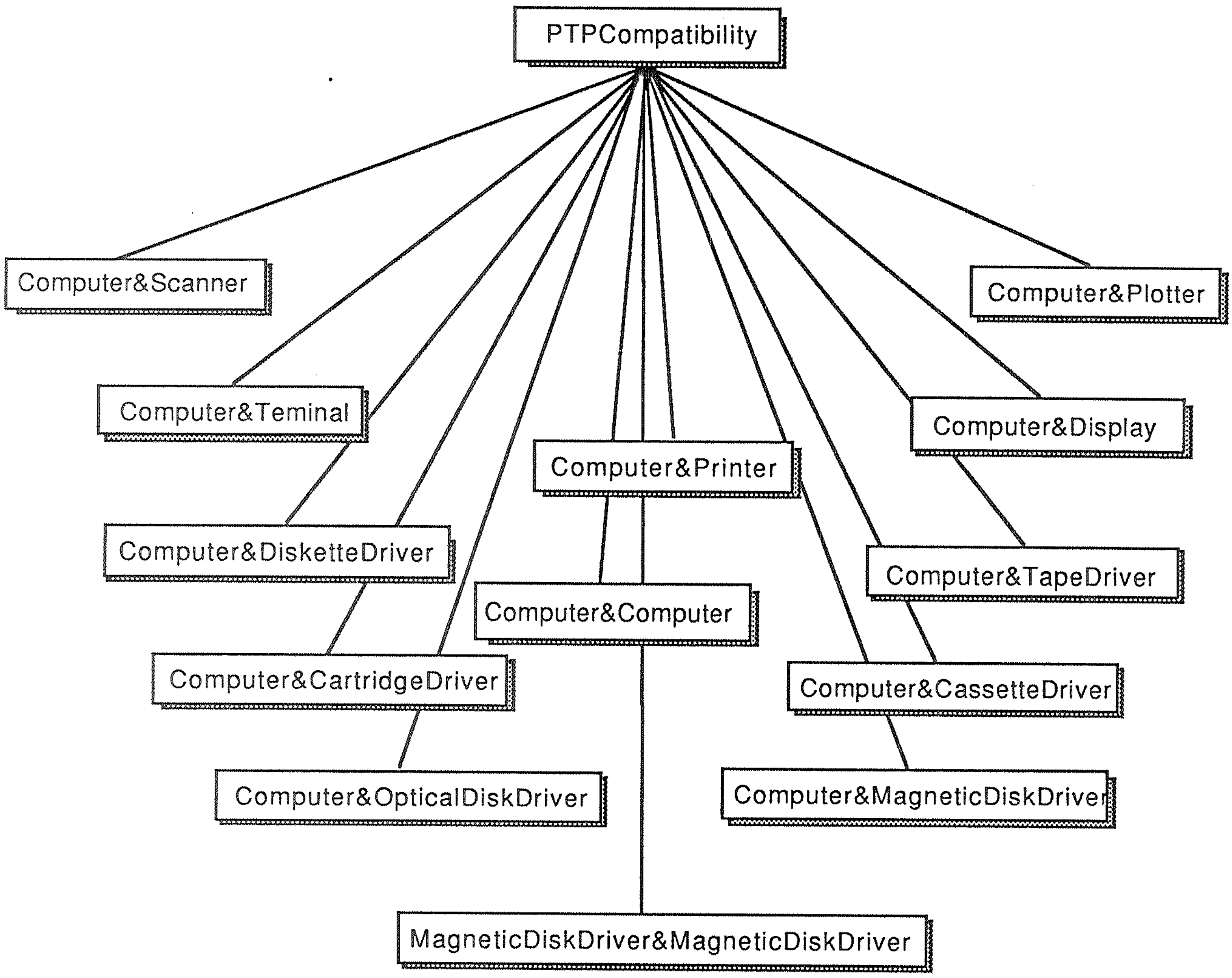

PTPCompatibility
Computer&Scanner
Computer&Plotter
Computer&Teminal
Computer&Display
Computer&Printer
Computer&DisketteDriver
Computer&TapeDriver
Computer&Computer
Computer&CartridgeDriver
Computer&CassetteDriver
Computer&OpticalDiskDriver
Computer&MagneticDiskDriver
MagneticDiskDriver&MagneticDiskDriver

Figure 6

Figure 6 shows the specializations of class "PTPCompatibility".

An instance of a point-to-point compatibility class represents the following fact: a certain device (given by the value of property "Device1" of the instance) can be point-to-point connected to another device (value of "Device2"), provided that the connection is established through an interface of the first device which is of the proper type ("Interface1") and an interface of the second device which is of the proper type ("Interface2").

As for expansion compatibility classes, the semantics of point-to-point compatibility classes is given by their to-test procedure, which is non-standard. In general, the to-test procedure of a point-to-point compatibility class receives as input two hardware devices and searches the class extension to see whether there exists an instance asserting the compatibility of them. In doing so, the procedure tests also if the given devices have still available the external interfaces of the type required for the connection. The to-test procedure of terminal/computer and printer/computer compatibility classes has a different behavior, due to the fact that enumerating all the compatibilities between such devices would be impractical, as their number is very high. In fact for a printer or a terminal to be compatible with a computer, it is sufficient that they have a common interface (which is usually a serial RS232 for a terminal and a parallel interface for a printer). Thus, unlike the other compatibility classes, terminal/computer and printer/computer compatibility classes have no instances: the to-test procedure ascertains the compatibility between two devices of these kinds just by checking whether the computer has available the external interface that matches with that of the terminal or printer.

5.4. Architectural Units

The most general architectural unit class is "ArchitecturalUnit", an instance of metaclass "ArchitecturalUnitClass". Figure 7 shows the specializations of "ArchitecturalUnit", which include the four classes that model the four levels of Architectures, plus the class "Expansion" (whose instances represent computer expansions), and class "AuxArchitecturalUnit", which plays a role analogous to that of "AuxCatalogue".

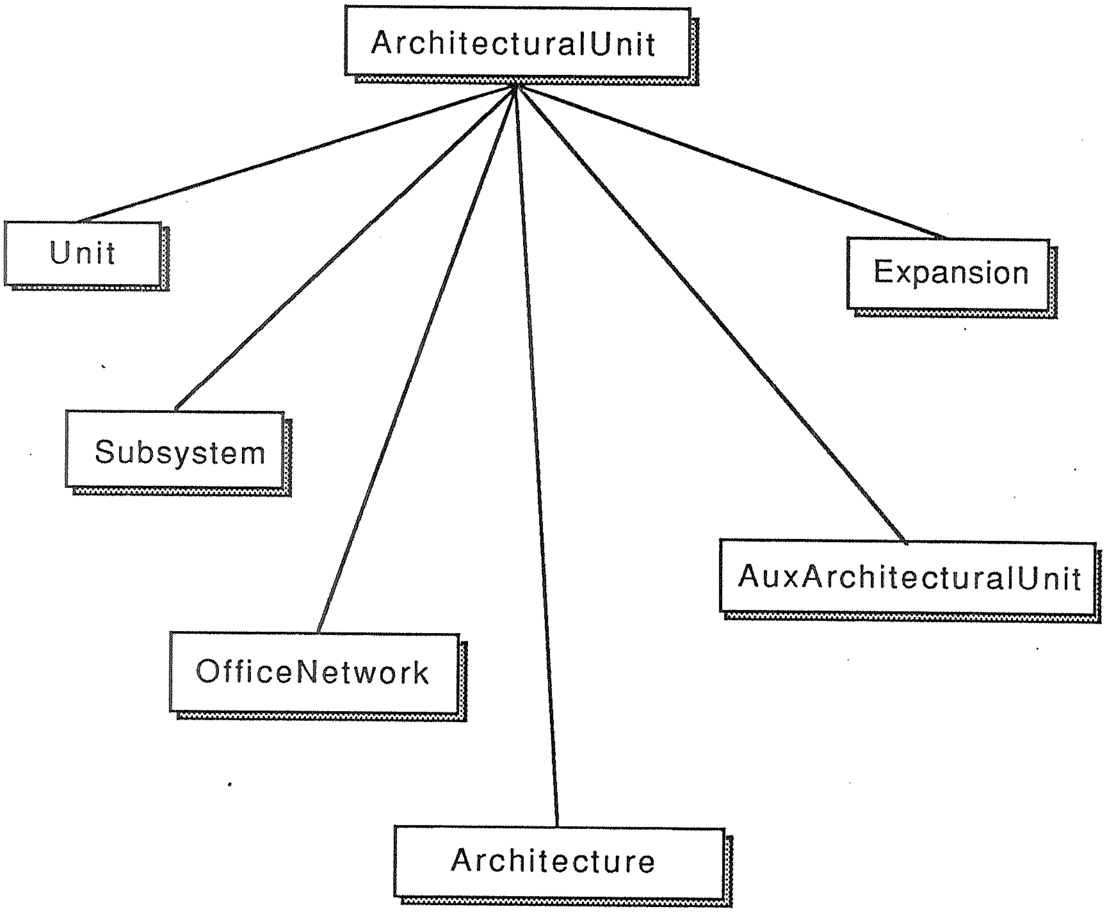

ArchitecturalUnit
Unit
Expansion
Subsystem
AuxArchitecturalUnit
OfficeNetwork
Architecture

Figure 7

5.4.1. Units

Units are instances of class "Unit", which corresponds to class "Component" of the Catalogue, as it represents materializations of instances of "Component". The correspondence between Catalogue objects and the Units that represent them in Architectures, is realized by making the is-a offspring of class "Unit" (shown in figure 8) isomorphic to that of "Component". Thus, "SoftwarePackage" corresponds to "SoftwareComponent", and has as instances objects that represent software components when they are used in Architectures. Likewise, class "HardwareUnit" corresponds to "HardwareComponent", and its specializations "StaticUnit" and "ExpandableUnit" are one-to-one with classes "StaticComponent" and "ExpandableComponent", respectively. As expected, "ExpandableUnit" is specialized by classes "ComputerUnit" and "PeripheralUnit". An instance of any of these classes is related to the corresponding Catalogue object through the property "BasicModel", defined by "Unit" and therefore inherited also by its specializations.

In the next two Sections we will describe the implementation of computer and peripheral Units, the details of the other Unit classes being not particularly relevant.

5.4.1.1. Computer Units

Computer Units are instances of class "ComputerUnit", which is further specialized in a way that mirrors the specialization of its corresponding Catalogue class, i.e. "ComputerT". "ComputerUnit" does not add any structural property to "BasicModel", but defines a number of assertional relationships. These can be divided into three groups. The first group consists of the relationships describing the Units that are part of the computer Unit:

- "InstalledDisplay", linking the computer Unit to its display Unit (if any), which is an instance of class "DisplayUnit";
- "InstalledCPU", the same for the computer Unit's CPU;
- "InstalledKeyboard", the computer Unit's keyboard, if any;
- "InstalledPointingDevice", the computer Unit's pointing

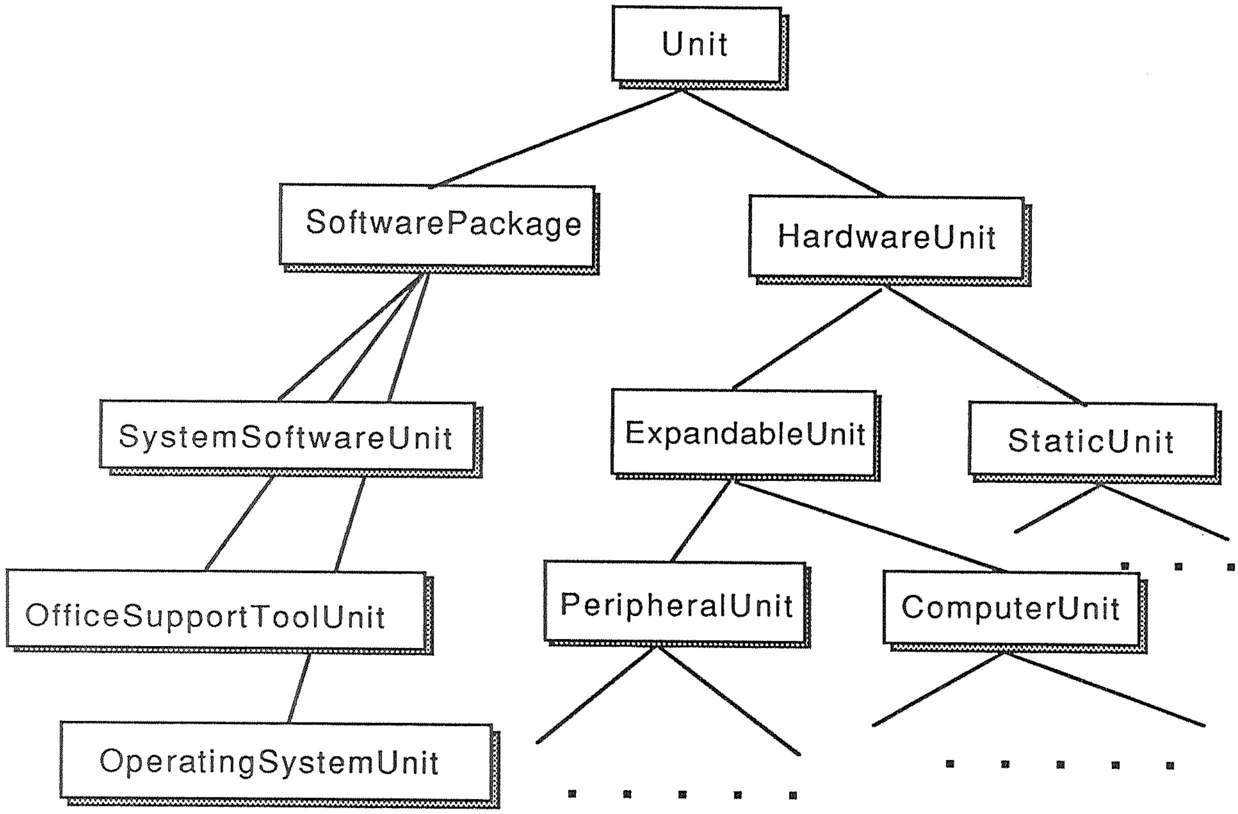

Unit
SoftwarePackage
HardwareUnit
SystemSoftwareUnit
ExpandableUnit
StaticUnit
OfficeSupportToolUnit
PeripheralUnit
ComputerUnit
OperatingSystemUnit

Figure 8

device, if any;

- "InstalledStoragePeripherals", the set of the peripheral Units of the computer, of all three kinds (sequential access, external random access, and internal random access);

- "InstalledUserMainMemory", associates to the computer Unit the quantity (in kilobytes) of main memory available for the user;

- "InstalledMainMemory", same as above, for the total main memory of the Unit.

The second group of relationships includes those representing the hardware machinery that the computer Unit makes available for being expanded or point-to-point connected to other Units:

- "FreeExternalInterfaces", which associates to the computer Unit the set of external interface Units for connecting point-to-point the Unit to other computer or peripheral Units; this relation is defined by "ExpandableUnit" and inherited by both "ComputerUnit" and "PeripheralUnit";

- "FreeNetworkInterfaces", relating computer Units to their available network interfaces Unit;

- "FreeExpansionSlots", links the computer Unit to the set of expansion slot Units made available by the Unit for hardware expansions.

Finally there is the third group of relationships asserting the expansions that have been performed on the computer Unit; these are:

- "SoftwareExpansion", representing the set of software expansions made to the computer Unit; software, as well as hardware expansion objects, are described in Section 5.4.2;

- "HardwareExpansions", same as above for hardware expansions;

- "AddedCoprocessors", whose values associated to the given computer Unit represents the coprocessor board Units installed on the Unit. This relationship is redundant, as

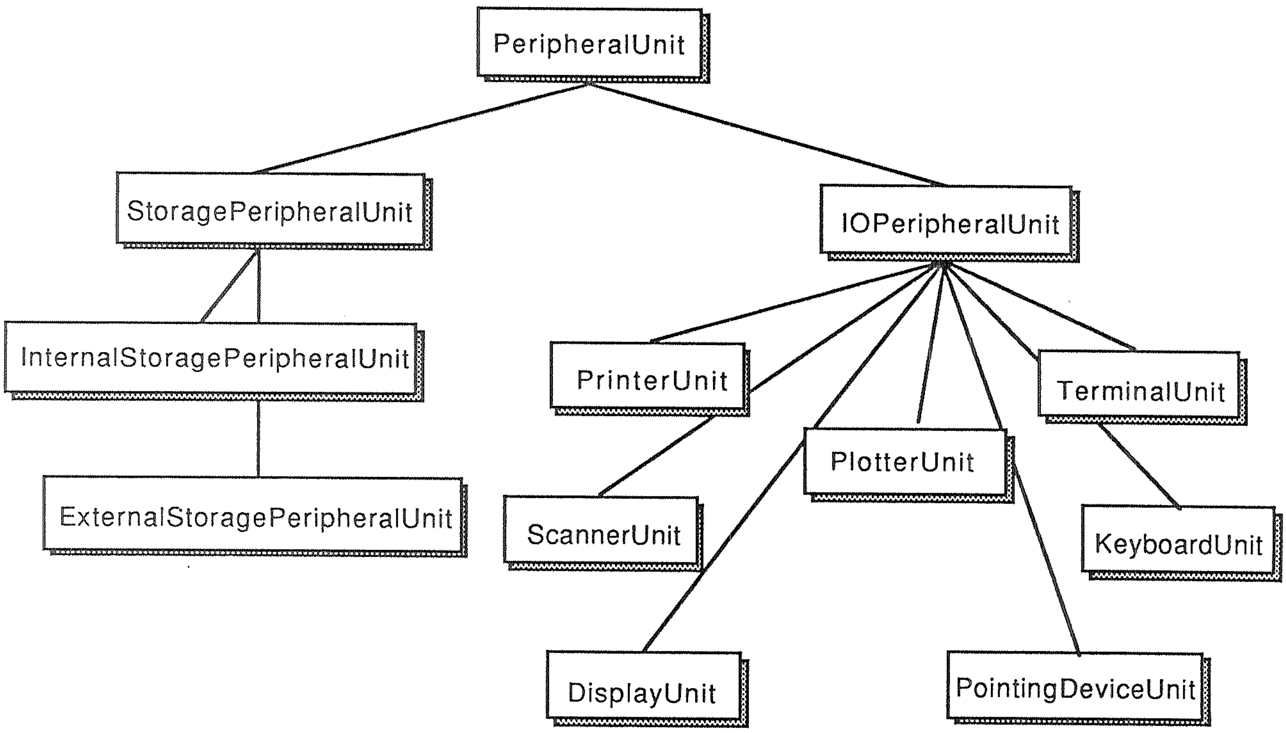

PeripheralUnit
StoragePeripheralUnit
IOPeripheralUnit
InternalStoragePeripheralUnit
ExternalStoragePeripheralUnit
PrinterUnit
TerminalUnit
PlotterUnit
ScannerUnit
KeyboardUnit
DisplayUnit
PointingDeviceUnit

Figure 9

coprocessor expansions are already described as hardware expansions; however, it has been included to make directly accessible from the computer Unit the set of CPUs it has.

When a computer Unit is created, the user only has to specify which computer of the Catalogue the Unit being created represents. It is the (non standard) to-put procedure associated to class "ComputerUnit" that takes the responsibility of providing the above relations with an initial value. This means also that the Units representing parts of the computer Unit (CPU, keyboard, pointing device, etc.) are created by the "ComputerUnit" to-put procedure, as a side effect of the creation of the computer Unit. Analogously, before effectively doing the removal, the to-rem procedure of "ComputerUnit" retracts the relations defined on the computer Unit to be removed, proceeding also to the part-of Units destruction.

5.4.1.2. Peripheral Units

The most general peripheral Unit class, "PeripheralUnit", is specialized isomorphically to the Catalogue class "Peripheral" (as it is shown in figure 9). Like "ComputerUnit", the only structural property of "PeripheralUnit" is "BasicModel", whereas of the relations defined for "ComputerUnit", only "FreeExternalInterfaces" also applies to peripheral Units, which cannot be expanded nor directly connected to a local area network. The to-put and to-rem procedures attached to class "PeripheralUnit" are analogous to those defined for "ComputerUnit".

5.4.2. Expansions

Expansion objects, instances of class "Expansion" (the most general expansion class), represent the extension of the capabilities of a computer Unit by means of expansion boards or software packages. Thus expansions may be of two kinds, hardware and software expansions, a fact that is realized in ASL by defining classes "HardwareExpansion" and "SoftwareExpansion" as direct specializations of "Expansion". The is-a offspring of class "Expansion" is illustrated in figure 10.

Hardware expansions are of the following kinds: main

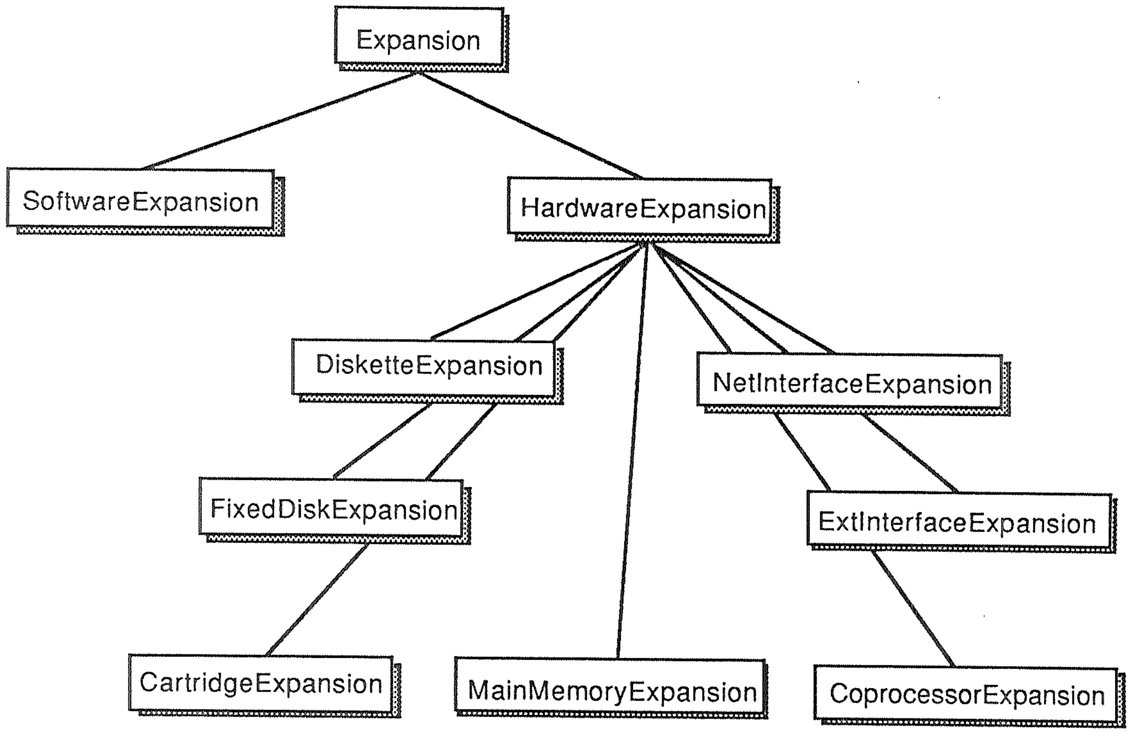

Expansion
SoftwareExpansion
HardwareExpansion
DisketteExpansion
NetInterfaceExpansion
FixedDiskExpansion
ExtInterfaceExpansion
CartridgeExpansion
MainMemoryExpansion
CoprocessorExpansion

Figure 10

memory, coprocessor, external interface, network interface, cartridge, fixed disk, and diskette expansions. All these expansions are modelled by defining an expansion class for each kind, and making it "is-a" "HardwareExpansion". They inherit from "HardwareExpansion" the properties "UsedExpansionSlot" and "UsedExpansionBoard", specializing them by properly refining the properties' range.

The creation of a hardware expansion is carried out by the to-put procedure associated to the corresponding class, and, with the exception of main memory expansions, it works as follows:

(1) the computer Unit to be expanded is checked to ascertain whether it has a free expansion slot Unit of the required kind;

(2) if yes, the appropriate expansion compatibility class is searched, in order to find out whether there exists a board satisfying the user request that can be installed on the Unit in question;

(3) if such a board is found, the expansion is made by accomplishing the following steps:

Step 1: a Unit representing the board is created;

Step 2: an instance of the appropriate expansion class is created; this instance has the board Unit as value of the property "UsedExpansionBoard", and the slot Unit found in (1) above as value of the property "UsedExpansionSlot";

Step 3: an instance of the relation "HardwareExpansions", having the expanded Unit as domain element, and the expansion instance as range element, is created, thus associating the expansion to the expanded Unit;

Step 4: the pair <expanded Unit, used slot> is removed from relation "FreeExpansionSlots", in order to represent the fact that the slot Unit of the expanded Unit that has been used for the expansion is no longer free;

Step 5: operations that are specific to the particular expansion being made are performed; for instance, in case of an external interface expansion, the external interfaces carried by the expansion board must be added to the set of free external interfaces of the expanded Unit.

For the reasons that have been pointed out in Section 5.2.2.4, main memory expansions are not realized in this way; in particular, as there are no main memory expansion slots, point (1) above is performed by testing whether the main memory installed on the computer Unit is equal to the maximum allowed, a value that can be found in the ASL Catalogue. This is the only difference between main memory and the other kinds of expansions; as it can be seen, it does not affect the generality of the modelization.

Software expansions are represented in an analogous way. They are instances of class "SoftwareExpansion", which has no specializations because the differences between the software packages that can be installed by a software expansion are less relevant than those between expansion boards. For the same reason, the class "SoftwareCompatibility" has no specializations. "SoftwareExpansion" defines two properties: "ComputerUnit", whose value gives the Unit which the expansion refers to, and "AddedSoftwareTool", which represents the software package employed in the expansion. The to-put procedure that gives the semantics to class "SoftwareExpansion" receives as input a computer Unit and a software package, and behaves as follows:

(1) it checks whether the specified software is compatible with the computer Unit, by testing class "SoftwareCompatibility";

(2) if yes, it makes the expansion by creating an appropriate instance of class "SoftwareExpansion", and adding a pair <expanded Unit, employed package> to relation "SoftwareExpansion".

Expansions, whether hardware or software, can also be removed, a task that is performed by the to-rem procedure associated to the involved expansion class. Besides removing the expansion instance, these procedures properly manipulate the involved relations defined for computer Units.

5.4.3. Subsystems

ASL Subsystems are instances of class "Subsystem", and may be of two kinds: simple Subsystems, consisting of only one computer Unit, or complex Subsystems, given by a computer Unit point-to-point connected with other expandable Units (i.e. computers or peripherals). This categorization of Subsystems is realized by making class "ComputerUnit" (and, by transitivity, all its specializations) "is-a" "Subsystem", and by defining a class, called "PTPConnectedSubsystem", which collects complex Subsystems and which is also a specialization of "Subsystem". Thus, the ASL interpreter 'knows' that computer Units are Subsystems, and this avoids to define twice simple Subsystems, once as Computer Units, and then as Subsystems.

Complex Subsystems may be viewed as undirected graphs, whose nodes are the Units that constitute the Subsystem, and whose arcs are the point-to-point connections between such Units. These graphs have a variable structure, due to the fact that the Units and the connections between them may change during the Subsystem lifetime. This situation is modelled by defining no properties for class "PTPConnectedSubsystem", but making this class the domain of two relationships:

- "PTPConnections": each pair of which links a complex Subsystem to one of its point-to-point connections, and

- "Units": each pair of which links a complex Subsystem to one of its Units.

As explained in Section 3.4, point-to-point connections are ASL objects, namely they are instances of class "PTPConnection", one of the auxiliary architectural unit classes. As it would be too heavy for the ASL user to create separately all the required point-to-point connections before creating a Subsystem, it is the procedure that creates complex Subsystems that takes this responsibility. This procedure is the to-put procedure associated to the class "PTPConnectedSubsystem"; it requires as input the list of Units that are to be included in the Subsystem, and a list of Unit pairs, each pair representing a point-to-point connection that must be established between two Units of the Subsystem. The procedure performs the following steps:

(1) it checks whether the input data are consistent; this involves a number of controls, ranging from the check on the type of the data to that on the congruency between the list of Units and the list of Unit pairs;

(2) it checks whether the specified Subsystem topology is correct, i.e. if the graph is serial (see Section 3.4);

(3) for each point-to-point connection to be established, it checks whether the Units to be connected are compatible and whether they have an available free external interface Unit;

(4) it establishes the point-to-point connections by creating appropriate instances of the class "PTPConnection";

(5) it creates the Subsystem.

Steps 3 requires a compatibility check for point-to-point connections. As explained earlier, this check is performed in a similar way to that on expansion compatibility. When the compatibility between two Units must be checked, the to-test procedure associated to class "PTPCompatibility" is invoked, and the two Units to be checked are passed to it. As already mentioned in Section 5.3.2, the to-test procedure for terminal and printer compatibilities are special; all the other compatibility testing procedures work as expected, that is by searching the extension of the appropriate compatibility class. If these Units are not compatible, the to-test procedure returns nil; otherwise, it returns the type of the interface that must be used for the connection on each Unit. This information is used in step 4, when the connections are in fact created. It may happen that more than one free external interface of some Units may be used for the connection, and the choice on which one to use may later affect the possibility of establishing other connections for the same Unit. An algorithm that finds the optimal usage of external interfaces is used to resolve this kind of conflict, thus ensuring the creation of the Subsystem whenever possible. To assert a point-to-point connection between two Units, an instance of "PTPConnection" is created with the following property values:

- "Unit1": the first Unit connected (the order between Units is unimportant);

- "Unit1Interface": the external interface Unit of the first Unit that is used for the connection;

- "Unit2": the second Unit connected;

- "Unit2Interface": the external interface Unit of the second Unit that is used for the connection.

The creation of the required point-to-point connections is then a side effect of the creation of the Subsystem, which is finally accomplished at step 5. All the objects whose creation is a side effect of the creation of a Subsystem, are automatically removed when the Subsystem is removed.

5.4.4. Networks

The class of all ASL Networks is "OfficeNetwork", a specialization of "ArchitecturalUnit", which is in turn specialized into "TreeNetwork", "RingNetwork", and "BusNetwork".

Only bus Networks have been implemented in the ASL interpreter being described. The only property defined by class "BusNetwork" is "CommunicationChannel", whose value gives the ASL Catalogue LAN that is used for the Office Network. Similarly to complex Subsystems, Office Networks have no properties describing the Network structure, which is time varying. Two relationships link an Office Network to its constituents:

- "Host": each pair of which represent the relationship between a Network and one of its Hosts;

- "NetTopology": each pair of which represent the relationship between a Network and a pair of its Hosts; the set of Hosts pairs related to a Network gives the topology of the Network, by describing the couples of adjacent hosts. This representation of topology can be also used for Ring and Tree Networks, except that in this last case each pair models a father-child relationship between Hosts.

A host of a Network is an instance of class "Host", an auxiliary architectural class. The class "Host" defines the following properties:

- "Subsystem": whose value gives the Subsystem that constitutes the Host;

- "EscapeUnit": whose value gives the computer Unit of the Host's Subsystem which directly connects to the Network;

- "EscapeInterface": which gives the network interface Unit that is used for connecting the escape Unit to the Network.

Thus a Host object is just a Subsystem with additional information describing how the Subsystem is connected to the Network.

Office Networks communicate between each other by sharing Subsystems. However, Hosts cannot be shared by different Networks. This implies that if the Subsystem s must be shared by Networks n1 and n2, then two Hosts, say h1 and h2, must be created, along with the tuples <n1, h1> and <n2, h2>, of relation "Host", which assert the membership of Hosts to Networks; the sharing of s is then represented by specifying the same value, that is s, for property "Subsystem" of both Hosts h1 and h2. This modelization realizes the distinction between the concepts of local host and of member of an Office Network, where the former is represented by an ASL Subsystem, and the latter by an ASL Host.

The creation of an Office Network object is performed by the to-put procedure associated to class "OfficeNetwork". It requires, among others, a check on the compatibility between each computer Unit (or escape Unit) that must be connected to the LAN and the LAN itself. This check is carried out by using the values of properties "RequiredNetInterface" and "NetSoftware" of class "LocalAreaNet", which give, respectively, the type of the network interface and the set of software packages which are required by the LAN. In particular, each escape Unit is checked to find out whether it has a free network interface Unit of the required type, and the appropriate software packages.

5.4.5. Architectures

Objects representing Architectures are instances of class "Architecture", the most general architectural class. The categorization of Architectures into simple (i.e. one Subsystem) and complex is realized by making class "Subsystem" a specialization of "Architecture", and defining an apposite class, called "ComplexArchitecture" whose instances represent complex Architectures, and which is "is-a" "Architecture". Thus, a simple Architecture is generated any time a Subsystem is created, whereas complex Architecture must be explicitly created.

The class "ComplexArchitecture" does not define any property, as the structure of an Architecture is time varying. Instead, it is the domain of two relationships:

- "Networks", each pair <a, n> of which represents the fact that the Office Network n belongs to (complex) Architecture a; and

- "Gates", which works similarly for Gates.

A complex Architecture is then seen as a set of Office Networks and a set of Gates, where Gates are instances of class "Gateway", a specialization of "AuxArchitecturalUnit". "Gateway" defines one property, "Gate", which ranges over class "Subsystem", and whose value gives, for a certain Gate, which Subsystem constitutes the Gate. A relation is defined to have class "Gateway" as domain, and that is relation "Nets". A pair <g, n> belongs to "Nets" if and only if the Subsystem s constituting the Gate g belongs to the Office Network n, and s is used to connect n to other Office Networks, where it must belong too. The concept of a Gate is thus similar to the concept of a Host: a Gate is a Subsystem with additional information about how the Subsystem is used to make Office Networks communicate. This concept is necessary because not necessarily a Subsystem which is shared by two different Networks is supposed to connect them. It must be explicitly declared as such, and an appropriate concept is needed to this end.

The creation of a complex Architecture does not require a substantial checking. The to-put procedure of class "ComplexArchitecture", which is in charge of this creation, receives as input a set of Networks, and controls that these Networks form a consistent (initial) configuration of the Architecture. This control is done by verifying that there exist a set of Gates connecting these Networks in such a way that there be no isolated Network. The so identified set of Gates is related to the Architecture being defined through the "Gates" relation, whereas the "Networks" relation is used to link the Architecture to the Networks that constitute it.

6. A Graphical Interface for ASL

It has already been explained that the ASL System provides two Interfaces, the Catalogue Administrator and the Architecture Designer Interface. These interfaces only differ for the operations that they make available, but their design and implementation criteria are the same, so we will not distinguish between them in this Section, where we will illustrate the main features of a graphical Interface developed for ASL. In particular, it will be described: (1) the basic environment made available to the user at the beginning of an ASL session; (2) the invocation ASL operations; (3) the visualization of the result of these operations; (4) the controls that are enforced on the operations at the interface level; (5) error handling; (6) the user buffer. Each of these topics will be discussed in a separate Section.

The Interface has been implemented as a C program that runs under the Unix Operating System, and that uses the graphical facilities provided by the SunView software library. It has been developed on a SUN 5/52 Workstation, with 4 Megabytes main memory and a 71 Megabytes hard disk. The major technical problem that has been encountered at the implementation level is the management of the communication between the various processes that are active during an ASL session. These processes are: the interface process, the ASL interpreter process (a Lisp process), and the processes associated to the windows currently opened, one for each window. The communication problems have been solved by using sockets, which are software tools that allow the exchange of messages between processes.

6.1. The ASL Window

The ASL Window is the window that appears on the screen at the beginning of an ASL Section, and it is shown in figure 11.

The ASL view of an Architecture as structured through five levels of abstraction, is made transparent to the ASL user by presenting him an ASL Window with five icons, which are one-to-one with the Catalogue, Unit, Subsystem, Network, and Architecture levels of ASL. As figure 11 shows, these icons are displayed at the top of the window, and the user

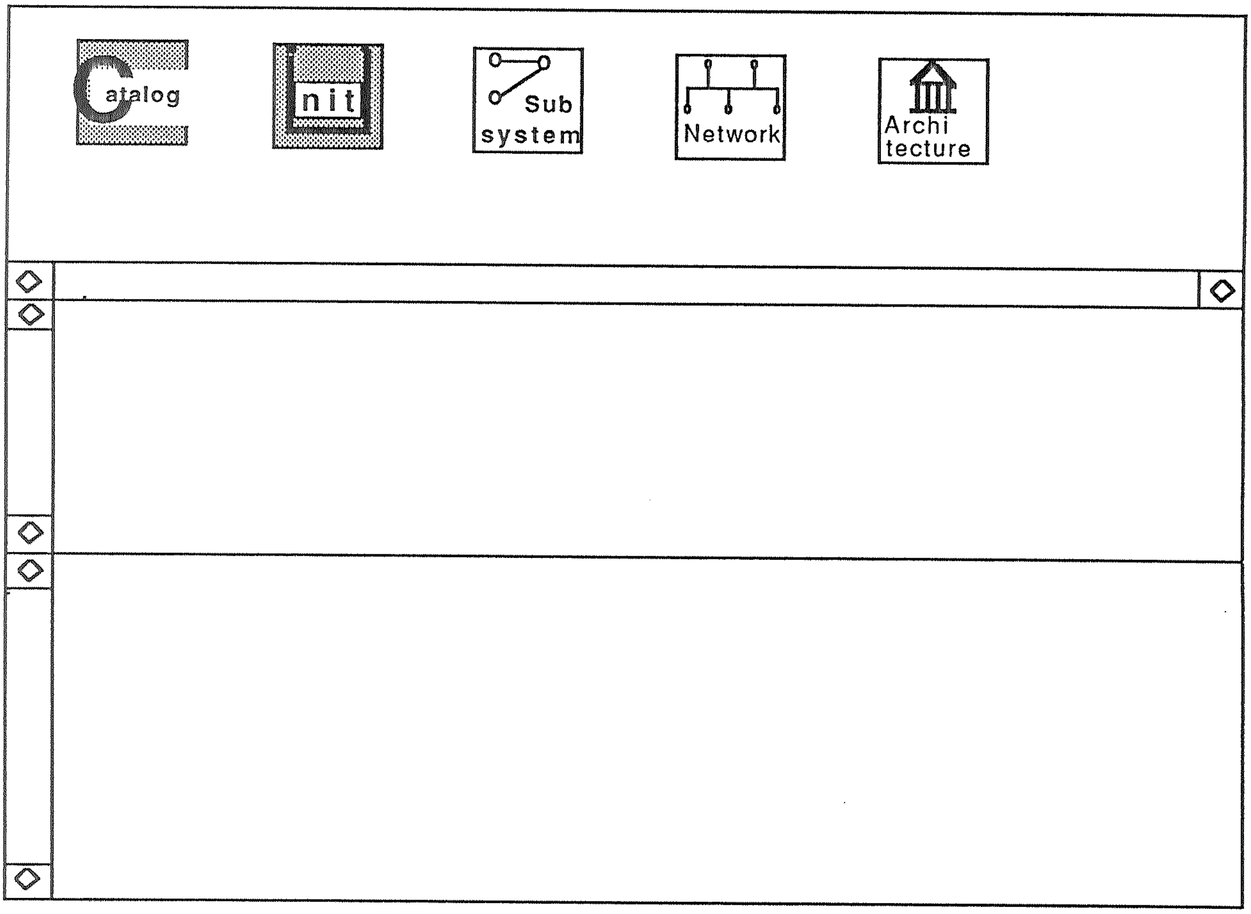


Figure 11

cannot move them around as normal Sunwindows icons. A menu is associated with each icon, called the Icon Menu. This menu presents the operations available at the ASL level corresponding to the icon, and can be obtained by positioning the cursor on the icon and clicking the right button of the mouse.

The space of the ASL Window is divided into three portions. In the topmost portion, the level icons are displayed at the beginning of a session. The other two portions are two windows: the window in the central part of the screen, called the Is-a Window, is used to display the Catalogue is-a hierarchy, which is very large and presumably very frequently consulted; the window in the bottom part of the ASL Window is a standard Text Subwindow, and is reserved for the user needs, thus it is called the User Window.

6.2. Invocation of the ASL Operations

The operations available at each ASL level can be viewed as hierarchically structured. For instance, the operations "create_component" and "create_version" of the Catalogue level, can be seen as specializations of a "create" operation, which is not a real ASL operation, but which can be considered an abstraction of the two. This operation hierarchy is realized by having a menu hierarchy at each ASL level. The root of the hierarchy is the Icon Menu associated to the level. In an Icon Menu, the name of each specialized operation is followed by an arrow: by following that arrow with the mouse, a menu containing the names of the specialized operations is displayed. This second level menu may contain in turn specialized operations, which are treated in the same way as in the Icon Menu. Figure 12.a shows the Icon Menu of the Catalogue level, and figure 12.b presents the second level menu corresponding to the "create" operation. This concept of a menu hierarchy is borrowed from the Sunwindow system, where it is known as 'walking menus'.

A window, called the Operation Window, is associated to each ASL operation. The Operation Window appears after its operation has been invoked via an appropriate menu selection. The name of the operation and the ASL level it belongs appear in the upper frame of the Operation Window. The Window is divided into two portions, whose usage will be explained later. It can be resized and moved around the screen.

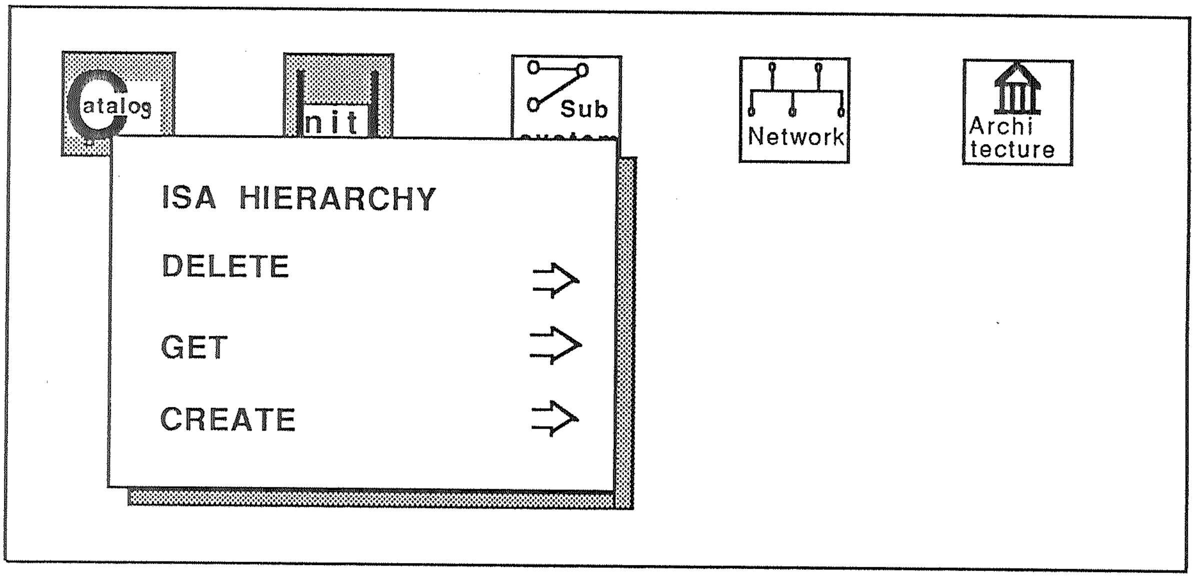


Figure 12.a

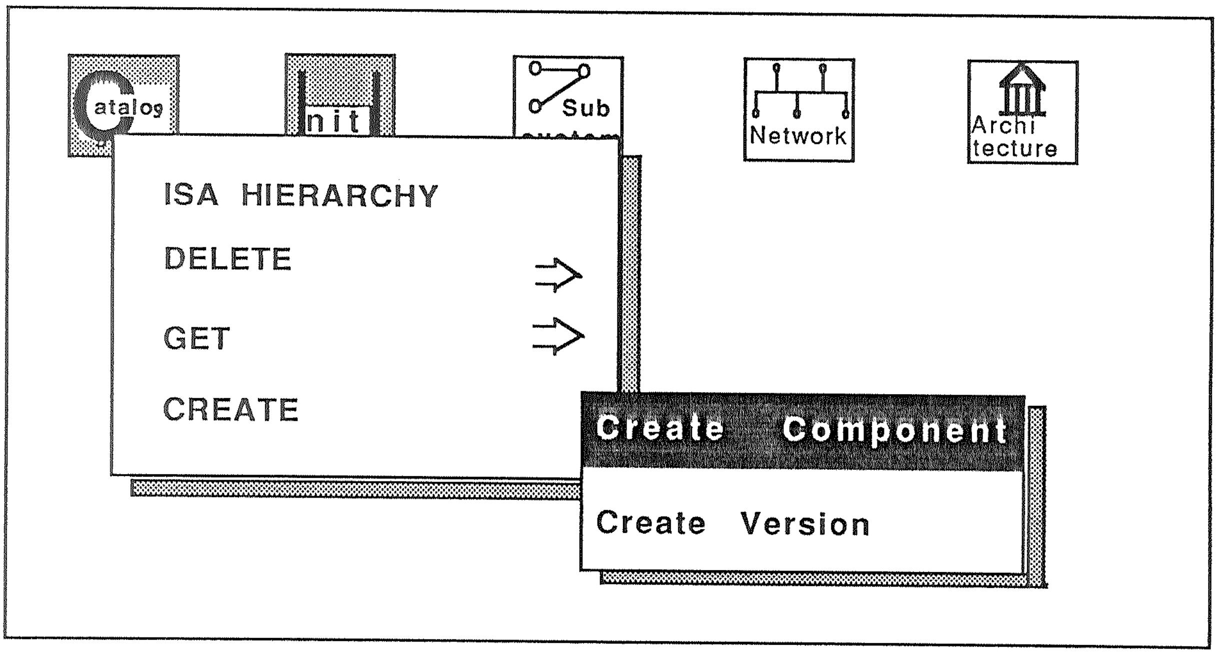


Figure 12.b

In almost all cases, the selection of an operation starts a dialogue between the user and the system, having as final aim the specification of an ASL operation. This dialogue takes place in the Operation Window, which therefore may have different forms depending on the current state of this dialogue. The user can talk to the system by selecting one of the buttons that appear in the window. Upon this selection, the content of the window is interpreted by the interface as input data for the execution of the action corresponding to the selected button. The execution of an action may result in a call to the ASL interpreter, in which case the dialogue has reached its last step and an ASL operation has been fired; or it may result in the display of a different form of the Operation Window, which means that further information are required to the user by the system.

As an example, let us consider the invokation of a "get_component" operation at the Catalogue level. Figure 13 shows the various steps of this invocation. In figure 13.a the selection from the second level menu is showed, whereas figure 13.b shows the initial form of the Operation Window. At this stage of the dialogue, the user is asked for the name of the class to be queried (that must be inserted after "ClassName"); after having typed this name the user can select one of the two buttons. Selecting "WITH CONDITION" (figure 13.c) he will get from the system the list of the properties of the specified class. He can successively fill the space left after each property with a predicate to be satisfied by the result of the query. This having done (figure 13.d), he can select the "EXEC" button, to finally fire the ASL operation that corresponds to the query. The result of the query, i.e. the list of qualifying objects, will be returned by the system in the Operation Window (figure 13.e).

The Operation Window does not automatically disappear after the execution of an operation. The user may remove the window by using the operations provided by Sunwindow. On the other hand, if he wishes to execute again the operation, he may go back to the desired state of the dialogue by using the "CLEAR" and "RESET" functions provided (where appropriate) by the interface to this end. In general, the "CLEAR" button restores the initial form of the Operation Window. In the "get_component" operation example, by clicking the "CLEAR" button after the display of the result, the user will get the Operation Window shown in figure 13.b. "RESET", instead, causes the last values specified by the user to be reset, so that new values can be given. For instance, after

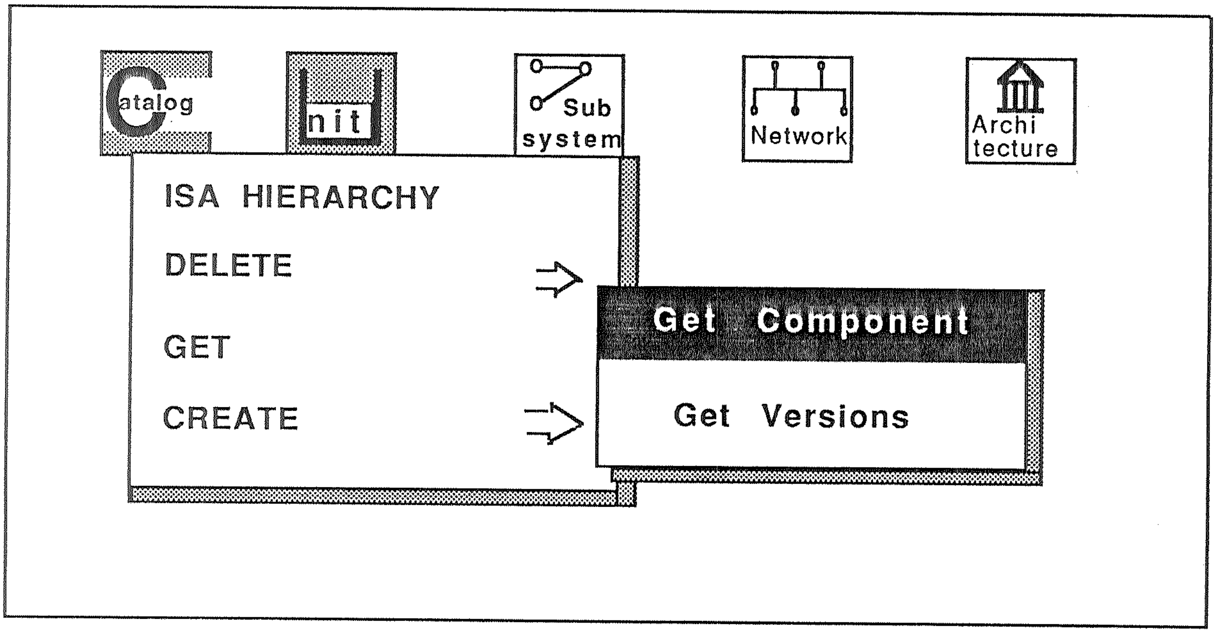

atalog
nit
Sub
system
Network
Archi
tecture
ISA HIERARCHY
DELETE
GET
CREATE
Get Component
Get Versions

Figure 13.a

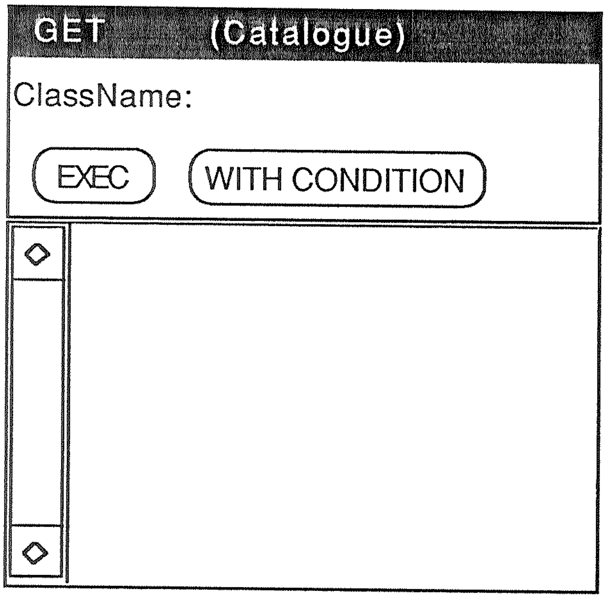


Figure 13.b

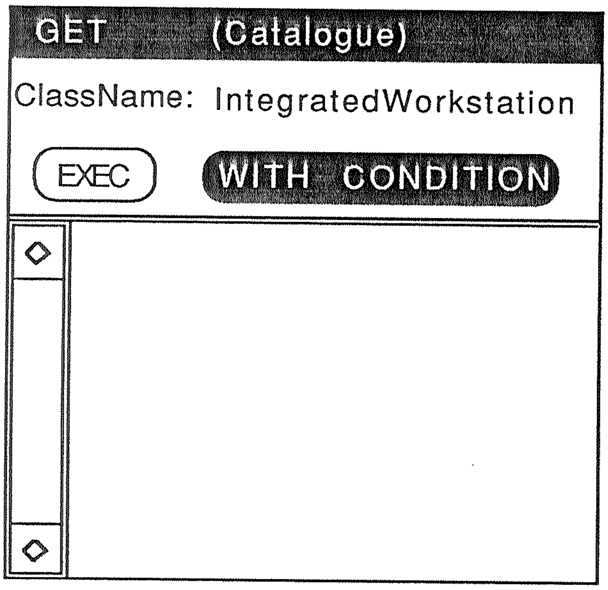


Figure 13.c

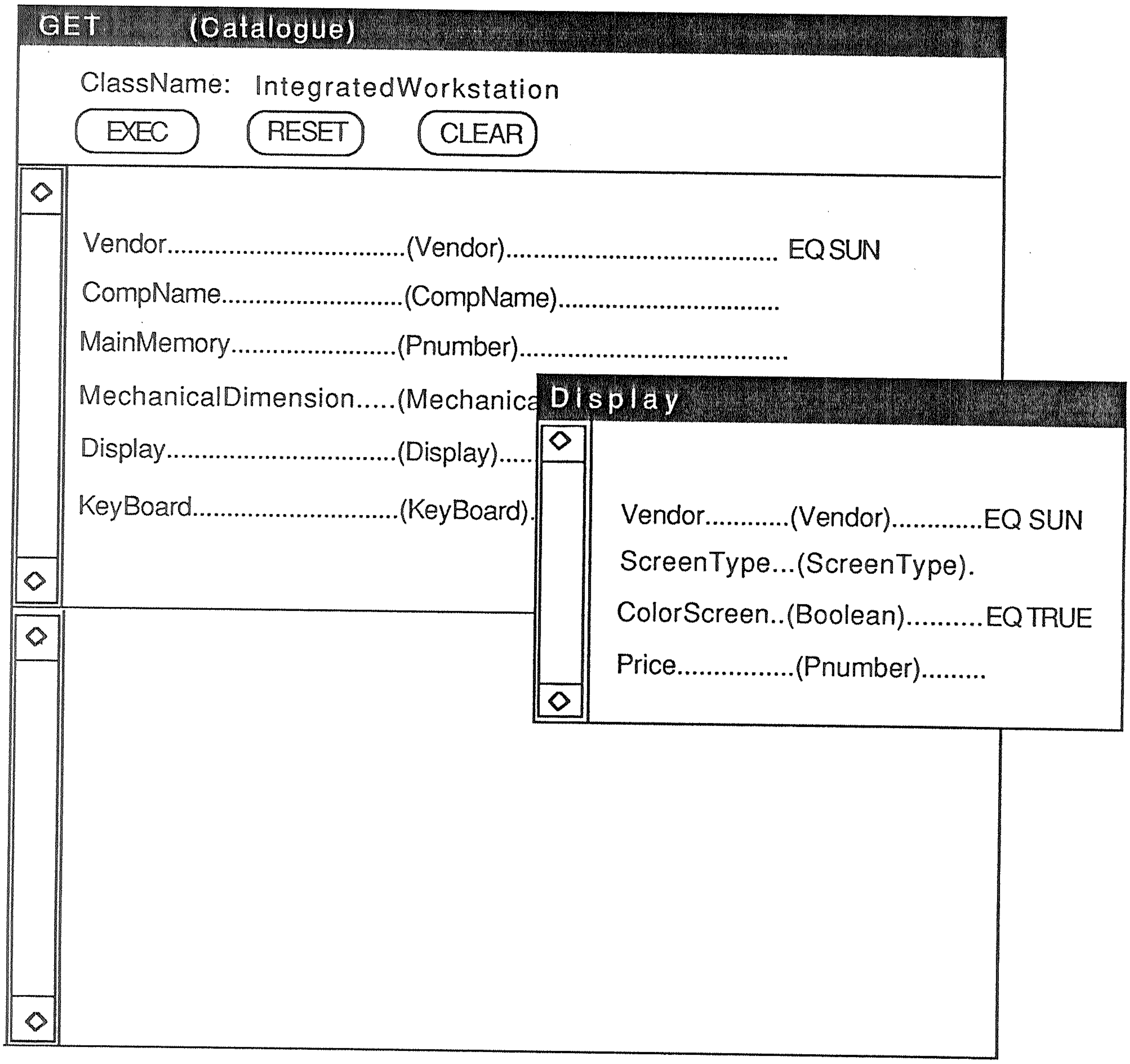

GET (Catalogue)
ClassName: IntegratedWorkstation
EXEC
RESET
CLEAR
Vendor..................................(Vendor)...................................... EQ SUN
CompName..........................(CompName)...............................
MainMemory........................(Pnumber).....................................
MechanicalDimension.....(Mechanica
Display...............................(Display).....
KeyBoard............................(KeyBoard).
Display
Vendor...........(Vendor).............EQ SUN
ScreenType...(ScreenType).
ColorScreen..(Boolean)..........EQ TRUE
Price................(Pnumber).........

Figure 13.d

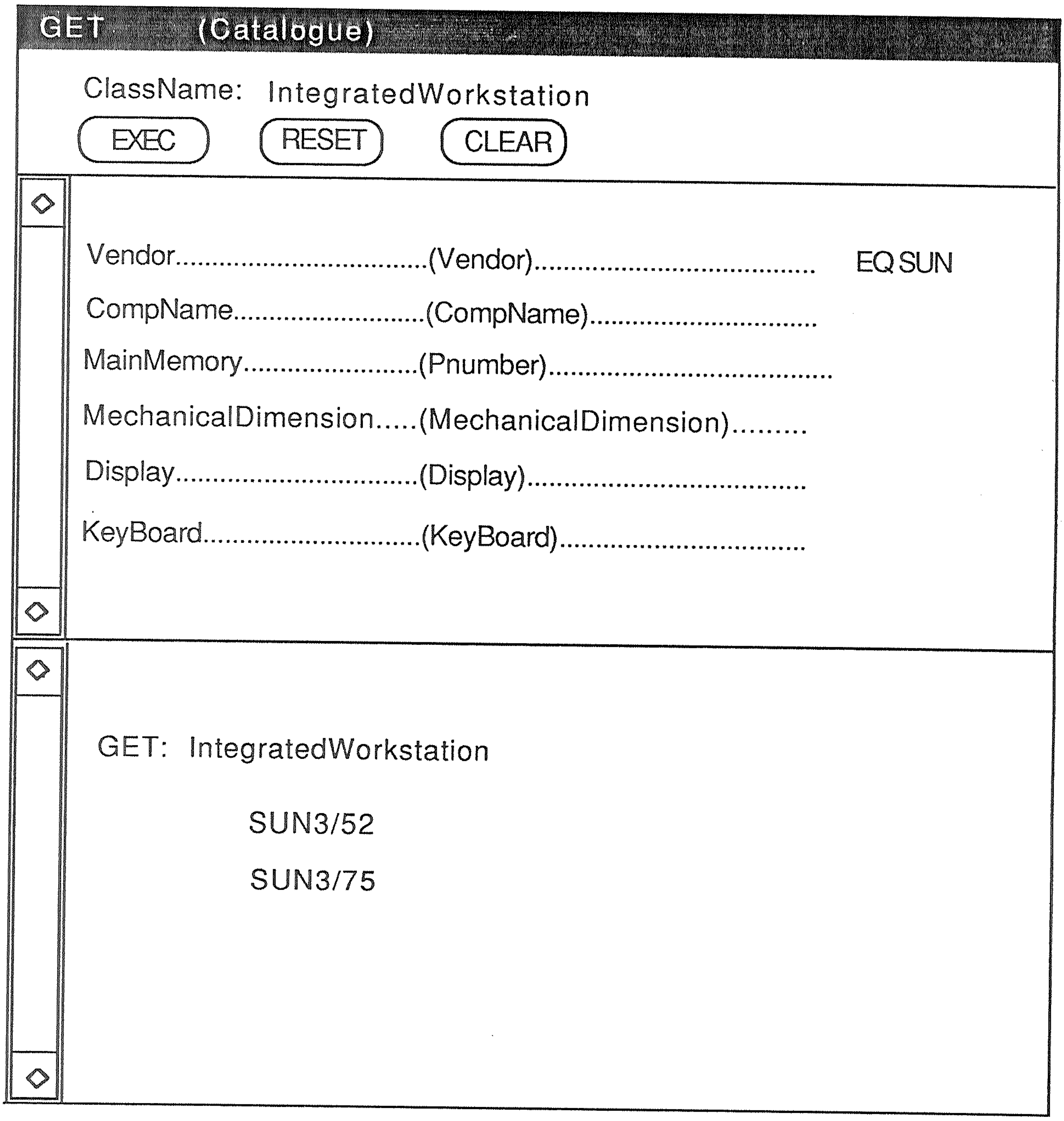

GET (Catalogue)
ClassName: IntegratedWorkstation
EXEC
RESET
CLEAR
Vendor.................................(Vendor)...................................... EQ SUN
CompName...........................(CompName)..............................
MainMemory.......................(Pnumber).....................................
MechanicalDimension.....(MechanicalDimension).........
Display................................(Display)....................................
KeyBoard............................(KeyBoard)................................
GET: IntegratedWorkstation
SUN3/52
SUN3/75

Figure 13.e

the creation of an object has been executed, the Operation Window still contains the pairs <property, value> specified for the object just created. By "RESET"-ing at this point, the values will be removed, so that the user can insert new values to create a new object.

6.3. Visualization of Results

ASL operations may in general have three kinds of results (excluding errors, which will be treated in Section 6.5):

(1) a 'yes' result, which typically comes from a creation or remove operation. This result is displayed by opening, in the middle of the screen, a small window containing the message that the operation has been successfully completed. This window can be removed by clicking any mouse button;

(2) a set of objects; this is the result of a query (with or without conditions) on an ASL class. The set of the returned objects is listed in the Operation Window from which the query has been issued (see figure 13.e);

(3) an object display. If the object is a token, its list of <property, value> pairs will be shown in a tabular form. If it is a class, the class definition will be given in tabula form, where each row of the table contains a pair <property, type>. In both cases the result will be returned to the user in the window where the operation has been specified. For Subsystem, Network, and Architecture objects, the result of the display may be given in graphical form. Figure 14 presents both the textual and the graphical display of a Subsystem consisting of four interconnected Units. In the topmost part of the Operation Window shown in figure 14, there are the buttons that can be used for talking to the system during the specification of the operation. The "EXEC" button serves to obtain the result after the name of the object to be displayed has been inserted after the "Subsystem-Name" string. The other two buttons can be used for choosing the display form. In the shown example, the "template" button has been used to get the textual display given in the middle window; then, the operation has been re-executed with the "graphic" button selected to get the graphical display shown in the bottom window.

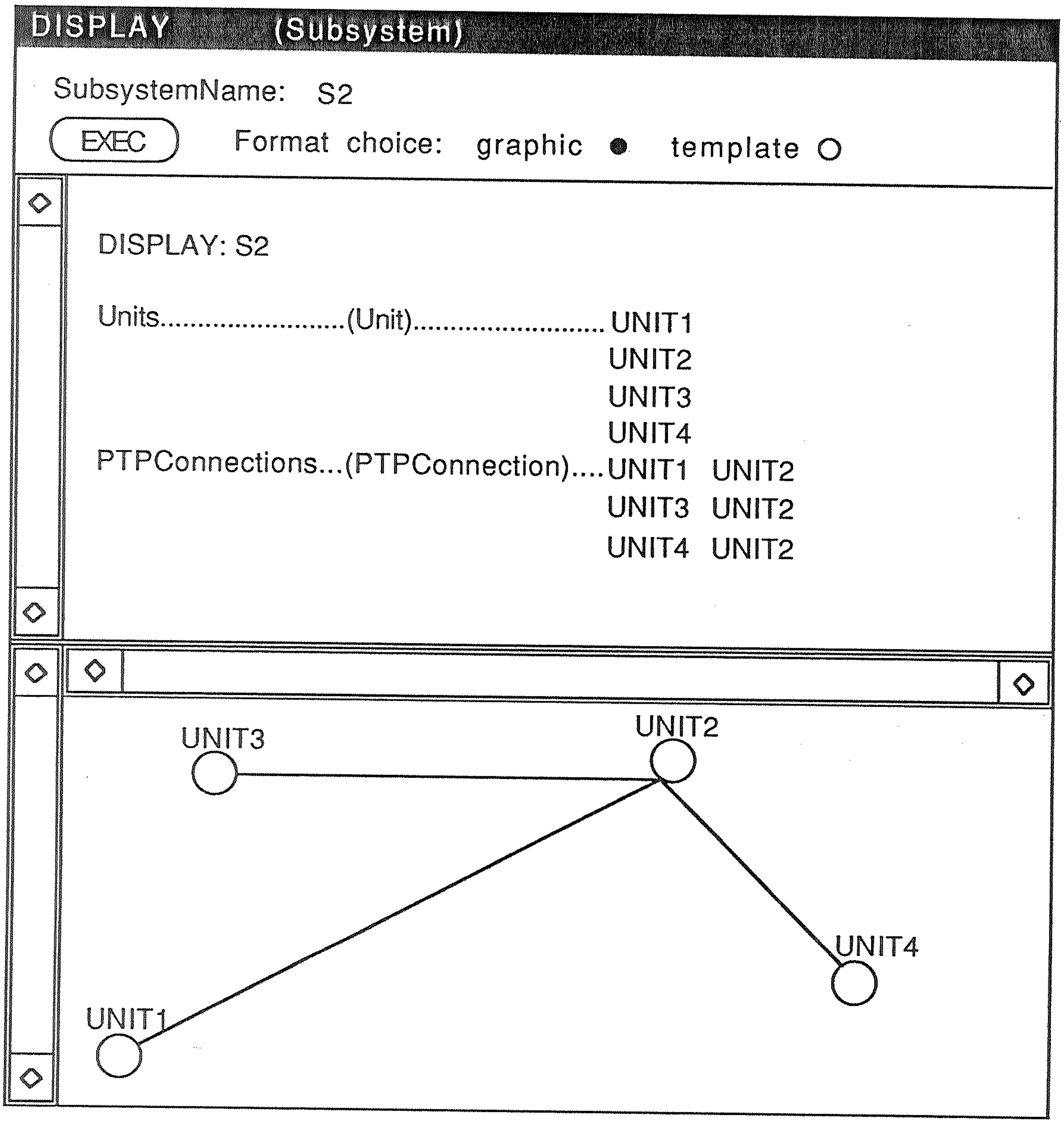

DISPLAY (Subsystem)
SubsystemName: S2
EXEC
Format choice: graphic template
DISPLAY: S2
Units........................(Unit).......................... UNIT1
UNIT2
UNIT3
UNIT4
PTPConnections...(PTPConnection)....UNIT1 UNIT2
UNIT3 UNIT2
UNIT4 UNIT2
UNIT3
UNIT2
UNIT4
UNIT1

Figure 14

A special treatement has been reserved to the display of the Catalogue is-a hierarchy. The lattice representing the is-a hierarchy is roughly eight times bigger than the whole screen. This makes it very inefficient to compute the lattice (or a part of it) each time the "is-a" operation is invoked. Moreover, if the is-a hierarchy were computed 'on the spot', a very sophisticated algorithm would have to be used in order to calculate the optimal display of the lattice on the screen, and this would be probably much more time consuming than the hierarchy calculation. For these reasons, the display of the is-a hierarchy is pre-computed and stored in a file which is loaded in the Is-a Window when the "is-a" operation is invoked. Of course, the Catalogue is-a hierarchy cannot be changed even by the Catalogue Administrator. The size of the Is-a Window can be rearranged by the user. To allow the user the inspection of the lattice, the Is-a Window has a horizontal and a vertical scroll bar.

6.4. Controls Enforced by the ASL Interface

The interaction between the user and the system during the specification of an operation presents a double advantage. The first advantage is that the system guides the specification of an operation, so that the user does not have to know in advance the parameters to be specified, their order, and their type. For instance, when issuing a query on a certain class, the system first asks the user the name of the class, then it gives the user the list of the properties on which a predicate can be given (as shown in figure 13). But there is another major advantage deriving from using this technique, and that is the automatic enforcement of certain constraints on the user operations. If the user interacted directly with the ASL interpreter, there would be nothing that guarantees that the operation be specified in the correct form. It would be the responsibility of the ASL interpreter to check the operation form, performing a usually high number of checks, which are tedious to code, and which make the execution of the operation inefficient because they are time consuming. The presence of a graphical interface between the user and the system, makes it possible to enforce many constraints when specifying the operation, as the interface may prevent the user from making certain mistakes. As an example of this, let us consider again the "get_component" operation. When the user is given the list of properties (as in figure 13.d), the only thing that he can do is to write something in the space left at the right end of each row. Thus, the ASL interpreter is guaranteed that the query predicate only involves the properties of the queried class, and does not have to check for this. The same applies in the creation of an object, with the system providing the user with the list of properties to be filled in, and the user cannot change the portion of the window where these properties are displayed.

It would be too long to enumerate, operation by operation, the controls that are automatically enforced by the interface. We stress here only the effectiveness of a graphical interface in alleviating the language interpreter task concerning the operation consistency checking.

6.5. Error Handling

There can be two kinds of errors that can occur in the ASL system. The first kind consists of the errors detected by the ASL interface. These errors result in a message which is displayed to the user in an 'ad hoc' window that appears in the middle of the screen. The rest of the screen is left unchanged so that the user can issue the correct operation just editing the incorrect specification. The window containing the error message may be removed by clicking any mouse button. Examples of this kind of errors are the missing of a parameter needed for executing an invoked operation (as shown in figure 15), or the request of executing an operation before the termination of the current execution.

The second kind of errors are those detected by the ASL interpreter, which passes to the interface an error message to be displayed to the user. In this case, the message may be shown to the user either in an 'ad hoc' window, or in the window associated to the operation that caused the error. An error of this kind is presented in figure 16, as the creation of a Subsystem with a Unit not having a free external interface Unit to establish a point-to-point connection.

6.6. The User Window

The User Window occupies the bottom part of the ASL Window, and its 'raison d'etre' is to provide the user of the ASL system with a private workspace. The User Window is a

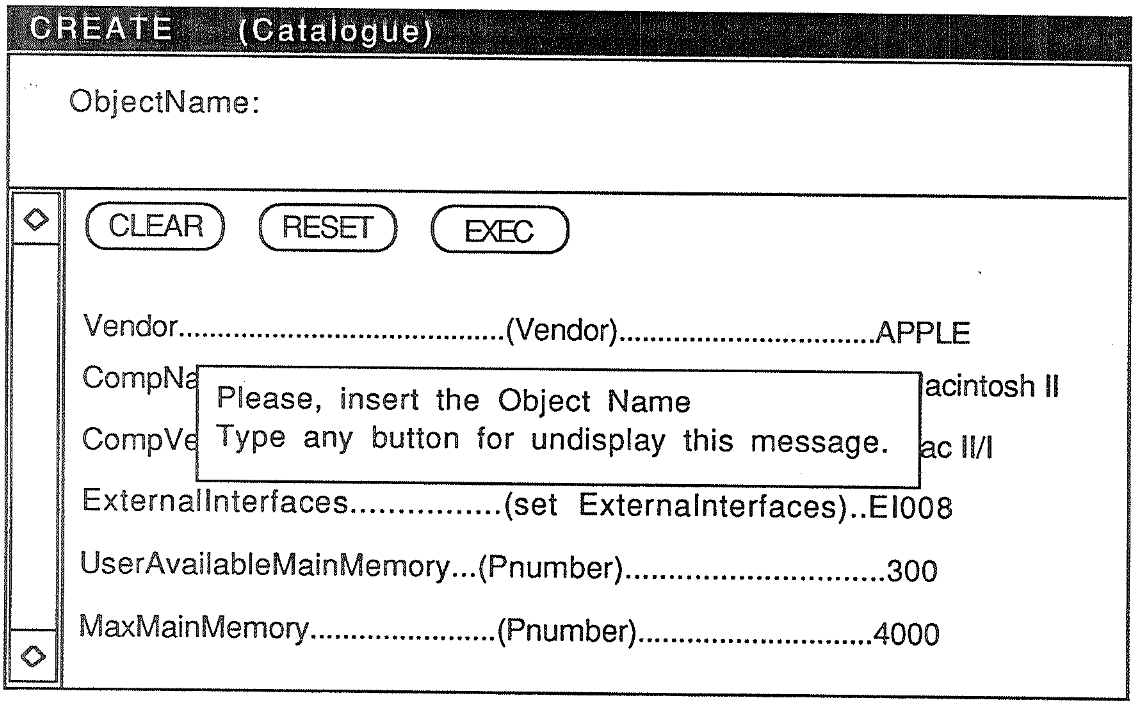

CREATE (Catalogue)
ObjectName:
CLEAR
RESET
EXEC
Vendor...(Vendor)................................APPLE
CompNa
acintosh II
CompVe
ac II/I
Please, insert the Object Name
Type any button for undisplay this message.
ExternalInterfaces................(set ExternaInterfaces)..EI008
UserAvailableMainMemory...(Pnumber)...............................300
MaxMainMemory.......................(Pnumber)............................4000

Figure 15

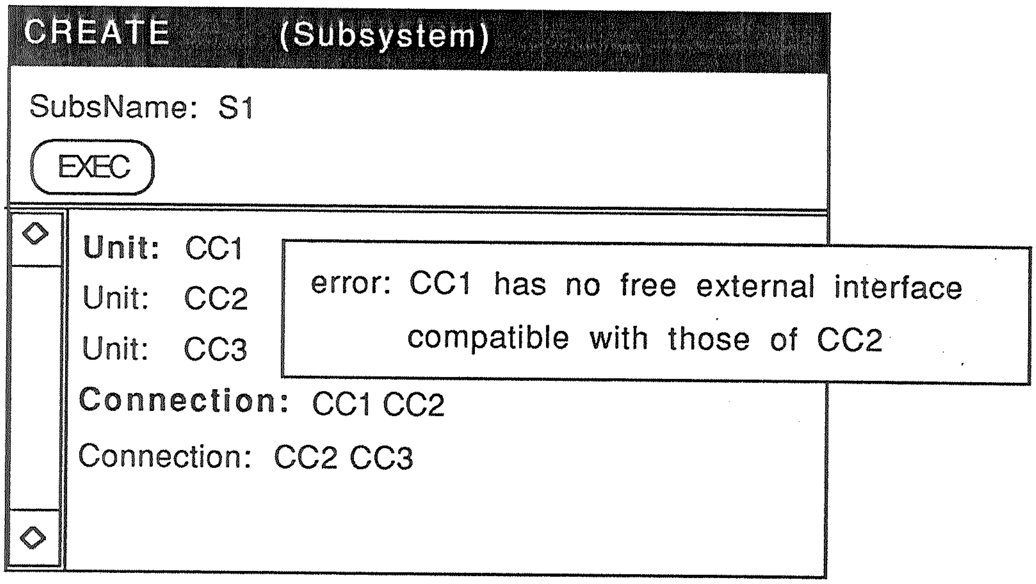

CREATE (Subsystem)
SubsName: S1
EXEC
Unit: CC1
Unit: CC2
Unit: CC3
Connection: CC1 CC2
Connection: CC2 CC3
error: CC1 has no free external interface compatible with those of CC2

Figure 16

standard Text Subwindow, i.e. a window for text editing that can be saved in a file, resized, and moved around the screen. Text can be selected by any other window and inserted in the User Window via the Sunwindow Selection Service, which relies on the mouse and a small set of functional keys of the SUN keyboard. Once the text has been put in the User Window, it can be processed via the normal editing facilities.

7. A Sample Insertion into the ASL Catalogue

In this Section we will show the insertion in the Catalogue of an object representing a personal computer, namely the IBM RT Personal Computer, Model 20. The source of the information that will be inserted is the IBM RT PC Hardware Maintenance and Service Manual [IBM].

A Catalogue insertion is a maintenance operation, which can be performed only by the Catalogue Administrator (CA). The ASL System provides the CA with a friendly graphical interface for operating on the Catalogue. For obvious typographical reasons, we can not illustrate the example through the CA graphical interface operations; therefore, we will directly use ASL operations, as they have been described in Section 4. The syntax of the operations that we will be presenting is close to that accepted by the ASL interpreter, except for few simplifications that have been made in order to improve the readability of this document. Briefly: the word "Create" introduces the creation of an object; the name of the class where the object being created belongs then follows, followed by the name of the object. Next, the word "with" introduces the (possibly empty) set of the object's properties. Each property consists of a name (the property name) and of an object name (the property value). The value of a multivalued property is denoted by a sequence of elements separated by blanks and enclosed in braces.

To make the exposition clearer, we begin by showing the operation that creates the object representing the computer; then we will describe the creation of the objects constituting the computer. Of course, this order must be reversed when talking to the ASL interpreter, as an object must first be defined to be used as property value of another object.

The object representing the IBM RT PC, named "PC1", will be an instance of class "PersonalComputer" and describe the basic configuration of the computer. It is created by the following operation, (where only non-nil property values have been included, and sets are represented by listing their elements between braces):

```
Create PersonalComputer PC1 with:
   Vendor: IBM;
   CompName: IBMRTPC;
   CompVersion: Model20;
   CPU: Intel386;
   MainMemory: 1000;
   UserAvailableMainMemory: 1000;
   MaxMainMemory: 8000;
   Display: Disp2;
   Keyboard: Keyb2;
   PointingDevice: PointingD2;
   TotalRAStorage: 41200;
   InternalRAStoragePeripherals: {DisketteD1, MDisk2};
   ExternalInterfaces: {EI24, EI25, EI26, EI27, EI28};
   NetworkInterfaces: {ANI2};
   ExpansionSlots: {CPUES1, CPUES2, DisketteES1, FixedDiskES1,
       FixedDiskES2, GenericES1, GenericES2, GenericES3,
       GenericES4, GenericES5, GenericES6, GenericES7};
   OperatingSystem: UNIXSV;
   AvailableLanguages: {CLang1, Fortran1};
   AvailableEditors: {Ed1, Emacs1};
   AvailableOfficeSupportTools: {Ingres1};
   SystemUnitDimensions: MeD6;
   Environment: Env4;
   PowerRequirements: PoR1;
end PersonalComputer PC1.
```

The first three properties are self-explanatory, and give general information on the computer model. The next properties (from "CPU" to "ExpansionSlots") describe the hardware characteristics of the computer. They are followed by properties (from "OperatingSystem" to "AvailableOfficeSupport-Tools") on the software packages that come with the basic configuration. The last three properties concern external parameters of the computer, namely the dimension of the system unit, environmental data and power requirements. Such properties will not be detailed, as not particularly interesting. In the next Section, we will concentrate on the hardware properties of the computer object; we will explain the value of such properties, and, when appropriate, show the creation of the objects that appear as values of those properties. We will do the same for software properties in Section 7.2. In the last Section, we will show the creation of some of the hardware and software components that are compatible with the IBM RT PC.

7.1. Hardware Definitions

The basic hardware configuration of the computer consists of the floor-standing system unit, a table top display, and a keyboard. In the ASL Catalogue, keyboards and displays are represented by instances of classes "Keyboard" and "Display", respectively. The association between a computer object and the appropriate display and keyboard objects is established by means of properties "Display" and "Keyboard" of class "PersonalComputer". The other hardware properties of "Computer" concern information about the system unit. In the next two Sections the definition of the IBM PC RT keyboard and display objects will be shown. In the remaining Sections we will describe the definition of the system unit, which has been subdivided into the following parts: processor, main memory, secondary storage, external interfaces, and expansion slots.

7.1.1. Display Definition

There are several displays that can be alternatively employed with the IBM RT PC being modelled. Among them, we choose as basic display the IBM 5151, which is also used on the IBM AT Personal Computer. The operation:

```
Create Display Disp2 with:
   Vendor: IBM;
   CompName: IBM5151;
   ColorScreen: false;
   GraphicScreen: false;
   ExternalInterfaces: {EI21};
end Display Disp2.
```

creates an instance of "Display", named "Disp2", whose properties represents (from the top down): the vendor of the display, its name, the facts that display "Disp2" has neither a color nor a graphic screen, and the set of the display's external interfaces. Object "EI21" is an external interface, created by the operation:

```
Create ExtInterface EI21 with:
   InterfaceType: DADPT2out;
end ExtInterface EI21.
```

where the type of "EI21" is a display adapter type, inserted in the Catalogue by the operation:

```
Create ParallelInterfaceType DADPT2out with:
   InterfaceName: DADPT2;
   Input/Output: O;
end ParallelInterfaceType DADPT2out.
```

The display connects to the system unit. This connection is not treated as a point-to-point connection, because it is not explicitly established by the user, but the same machinery used for point-to-point connection is employed for making it. In fact, when a Unit having "PC1" as basic model is created, the external interface Units that serve for the connection between the display and the computer are not included in the set of "FreeExternalInterfaces" of both Units, thus 'simulating' the creation of a point-to-point connection. In order to know which external interfaces must be used for this connection, the "Computer&Display" compatibility class is queried. In our case, the following compatibility instance will give us the information needed:

```
Create Computer&Display CDSComp5 with:
   Device1: PC1;
   Device2: Disp2;
   Interface1: DADPT2in;
   Interface2: DADPT2out;
end Computer&Display CDSComp5.
```

It asserts that a computer Unit whose basic model is "PC1" can be connected to a display Unit whose basic model is "Disp2", provided that an interface Unit whose basic model's type is "DADPT2in" is used for the computer Unit, and an interface Unit whose basic model's type is "DADPT2out" is used for the display Unit. The interface type "DADPT2in" is created in the same way than "DADPT2out", with the difference that it has "I" (input) as value of the property "Input/Output":

```
Create ParallelInterfaceType DADPT2in with:
   InterfaceName: DADPT2;
   Input/Output: I;
end ParallelInterfaceType DADPT2in.
```

An interface of this type is then included in the set of external interfaces of "PC1", as showed in Section 7.1.6.

The other display objects that can replace "Disp2" in the computer being described are defined in an analogous way, and used to define versions of the object representing the computer.

7.1.2. Keyboard and Mouse Definition

There is only one keyboard that can be used with the IBM RT PC. This keyboard has 102 keys, and connects directly to the system unit. This connection, like the system unit-display connection, is not explicitly described in the Catalogue.

The operation:

```
Create Keyboard Keyb2 with:
   Vendor: IBM;
   CompName: IBMKBD;
   NumbKeyboardKeys: 102;
   ExternalInterfaces: {EI22};
end Keyboard Keyb2.
```

creates the keyboard object associated to "PC1" by property "Keyboard". Analogously to the display, "EI22" is an external interface defined as:

```
Create ExtInterface EI22 with:
   InterfaceType: KADPT2out;
end ExtInterface EI22.
```

and whose type is given by:

```
Create ParallelInterfaceType KADPT2out with:
   InterfaceName: KADPT2;
   Input/Output: O;
end ParallelInterfaceType KADPT2out.
```

The connection between "PC1" and "Keyb2" Units is established as explained before, except that a computer-keyboard compatibility class does not exist, as this kind of connection is not included in the point-to-point connections allowed by ASL. So, the information on which interfaces must be used is obtained by matching the names of the interface types available on "PC1" and "Keyb2". This implies that at least one of the interfaces of "PC1" must be of a type whose name is "KADPT2", and this is in fact the case of adapter type "KADPT2in":

```
Create ParallelInterfaceType KADPT2in with
   InterfaceName: KADPT2;
   Input/Output: I;
end ParallelInterfaceType KADPT2in.
```

A Mouse is also available in the basic configuration. The corresponding Catalogue object is defined as follows:

```
Create PointingDevice PointingD2 with:
   Vendor: IBM;
   PointingDevType: Mouse;
   NumbPointingDeviceKeys: 2;
   ExternalInterfaces: {EI23};
end PointingDevice PointingD2.
```

The property "PointingDevType" describes the kind of the pointing device being defined; the ASL Catalogue 'knows' about three kinds of pointing devices: Mouse, Puck, and Tablet. The following adapter objects enable the connection between "PC1" and "PointingD2" Units, in the same way as for keyboards.

```
Create ExtInterface EI23 with:
   InterfaceType: MADPT2out;
end ExtInterface EI23.

Create ParallelInterfaceType MADPT2out with:
   InterfaceName: MADPT2;
   Input/Output: O;
end ParallelInterfaceType MADPT2out.

Create ParallelInterfaceType MADPT2in with:
   InterfaceName: MADPT2;
   Input/Output: I;
end ParallelInterfaceType MADPT2out.
```

7.1.3. Processor

In the basic IBM RT PC configuration, there is one processor board (in slot A), containing the 32-bit processor, the system memory controller, and the ROM modules. This board will be represented in the ASL Catalogue as an instance of class "CPU", defined in the following way:

```
Create CPU Intel386.
```

No properties are defined for a CPU object; although it might be useful to know some performance parameters of a CPU, these parameters are usually not provided by vendors, and very hard to derive by simulation. The property "CPU" links a computer object to the appropriate CPU object.

The property "BuiltInCoprocessors", also defined for computers, is used to associate a computer with its set of built-in coprocessors. As there is no built-in coprocessor in the basic configuration of the IBM RT PC, object "PC1" has the empty set (represented by the constant "nil") as value of property "BuiltInCoprocessors".

The IBM RT PC has two processor expansion slots that can be used to augment the processing capabilities of the computer. In the first one (slot 8), another processor board (identical to the built-in processor board) may be installed, whereas a floating-point board may be placed in slot B. These two expansion slots are defined by the following operations:

```
Create CoprocessorExpansionSlot CPUES1.

Create CoprocessorExpansionSlot CPUES2.
```

and linked to our computer through the "ExpansionSlots" property.

The floating-point processor board that can be installed on the IBM RT PC is created by the following operation:

```
Create CoprocessorBoard CPB1 with:
   Model: IBMRTCoprocessor;
   Functionality: Numerical
end CoprocessorBoard CPB1.
```

Also the standard processor board must be declared as an instance of class "CoprocessorBoard" in order to be used as an expansion board:

```
Create CoprocessorBoard CPB2 with:
   Model: Intel386;
   Functionality: Standard
end CoprocessorBoard CPB2.
```

In the definition of "CPB1", the "Model" property has as value a newly created instance of "CPU" ("IBMRTCoprocessor"), whereas the same property is valued "Intel386" in the definition of "CPB2" to signify that the standard IBM RT CP can also be used for an expansion. Now, to assert that "CPB1" and "CPB2" are expansion boards of "PC1", the following two instances of the computer-coprocessor compatibility class must be created:

```
Create Computer&CoprocessorBoard CCPComp1 with:
   Computer: PC1;
   Board: CPB1;
   Slots: {CPUES1}
end Computer&CoprocessorBoard CCPComp1.

Create Computer&CoprocessorBoard CCPComp2 with:
   Computer: PC1;
   Board: CPB2;
   Slots: {CPUES2}
end Computer&CoprocessorBoard CCPComp2.
```

The first two properties of these objects serve to relate "PC1" with an appropriate coprocessor expansion board, whereas the third property tells which slot of "PC1" must be used when performing the expansion. "Slots" is a multivalued property because, as it will be shown later, an expansion board may be alternatively installed in a set of expansion slots. Such set is then given as "Slots" property value.

7.1.4. Memory

The IBM RT PC System Unit has two slots (C and D) for main memory boards. There are three different kinds of board that can be placed in these slots; the boards differ from each other in the number of megabytes; they can be 1, 2, or 4 megabytes boards. Only nine different combinations of boards are possible, as shown by the following table:

Total Bytes of System Memory	Option in Slot C	Option in Slot D
1 MByte	1 MByte	-
2 MBytes	1 MByte	1 MByte
2 MBytes	2 MBytes	-
3 MBytes	2 MBytes	1 MByte
4 MBytes	2 MBytes	2 MBytes
4 MBytes	4 MBytes	-
5 MBytes	4 MBytes	1 MBytes
6 MBytes	4 MBytes	2 MBytes
8 MBytes	4 MBytes	4 MBytes

This situation is modelled in the Catalogue by assigning to the object representing the computer a main memory of 1 megabyte, which is the minimal option. The main memory of a computer object is represented via the property

"MainMemory", whose value gives (in kilobytes) the quantity of main memory installed in the basic configuration of the computer. Thus "PC1" has a value of 1000 for the property "MainMemory". The property "MaxMainMemory" represents the maximum main memory that can be supported by a computer. We have given a value of 8000 to this property in the object representing our computer. The last property concerning the main memory of a computer is the property "UserAvailableMainMemory", whose value give the main memory effectively available to the users. When the computer object is created, this property gets the same value as the "MainMemory" property. The installation of additional software packages may lower this value.

The remaining main memory options are described by defining six memory expansion boards associated to the computer object. Each board represents one of the six possible values of total megabytes of system memory, i.e. 2, 3, 4, 5, 6, and 8 total megabytes. By this modelization, the one-to-one correspondence between real world and model objects is lost, as the expansion board objects do not represent real expansion boards. However, since the ultimate goal of ASL is to describe architectures in order to measure their performance, we are only interested in knowing the total main memory of a computer, regardless the internal configuration that realizes such memory. The operation:

```
Create MainMemoryBoard MMEB1 with:
   Vendor: IBM;
   CompName: RTPC2MBMemoryExpansionOption;
   Dimension: 2000;
end MainMemoryBoard MMEB1.
```

creates the expansion board which, when installed as expansion to a computer, brings the total main memory of the computer to 2 megabytes. The definitions of the other five main memory expansion boards are similar and are not given.

To express the fact that this expansion board is usable within "PC1", we must create an appropriate instance of the compatibility class "Computer&MainMemoryBoard", as follows:

```
Create Computer&MainMemoryBoard CMMComp1 with:
   Computer: PC1;
   Board: MMEB1;
end Computer&MainMemoryBoard.
```

The compatibility between "PC1" and the other main memory boards is established in an analogous way.

7.1.5. Secondary Storage

Secondary storage devices are divided in the ASL Catalogue into four main classes: internal and external random-access, and internal and external sequential-access devices. The IBM RT PC system unit has neither external secondary storage devices, nor built-in internal sequential-access devices, whereas it has five drive positions (drive position A to E) for internal random-access secondary storage devices; in particular, drive positions C, D, and E are for fixed-disk drives, whereas positions A and B are for diskette drives.

Fixed-Disk Drives

In the basic configuration, the system unit always has a fixed-disk drives installed in position C. Drive positions D and E are optional drive positions. There are two fixed-disk drives that can be installed, in any combination, in the drive positions: the Type R40 and the Type R70. Their corresponding Catalogue objects are defined by means of the following operations:

```
Create MagneticDiskDriver MDisk2 with:
   Vendor: IBM;
   CompName: R40;
   FormattedStorageCapability: 40000;
   AccessType: ReadWrite;
end MagneticDiskDriver MDisk2.

Create MagneticDiskDriver MDisk3 with:
   Vendor: IBM;
   CompName: R70;
   FormattedStorageCapability: 70000;
   AccessType: ReadWrite;
end MagneticDiskDriver MDisk3.
```

The value of property "FormattedStorageCapability" gives the disk storage capacity in kilobytes. The smallest between the two disks ("MDisk2") is declared to be in the basic configuration, by having inserted it into the property "BuiltIn-RAStoragePeripherals" value of "PC1". "MDisk3" is instead used to define versions of "PC1". Drive positions D and E are defined as fixed disk expansion slots, by creating the following instances of class "FixedDiskExpansionSlot":

```
Create FixedDiskExpansionSlot FixedDiskES1.

Create FixedDiskExpansionSlot FixedDiskES2.
```

To enable the use of "MDisk2" and "MDisk3" also as expansion boards of type fixed-disk, two instances of class "Fixed-DiskBoard" must be created:

```
Create FixedDiskBoard FixedDiskB1 with:
   AllowedFixedDiskDriver: MDisk2;
end FixedDiskBoard FixedDiskB1.

Create FixedDiskBoard FixedDiskB2 with:
   AllowedFixedDiskDriver: MDisk3
end FixedDiskBoard FixedDiskB2.
```

Each board corresponds to one fixed-disk, the correspondence being established through the property "AllowedFixedDiskDriver"of the board object. The links between these expansion boards and the computer to which they apply are typically compatibility assertions, established by the following instances of the computer-disk compatibility class:

```
Create Computer&FixedDiskBoard CFDComp1 with:
   Computer: PC1;
   Board: FixedDiskB1;
   Slots: {FixedDiskES1}
end Computer&FixedDiskBoard CFDComp1.

Create Computer&FixedDiskBoard CFDComp2 with:
   Computer PC1;
   Board FixedDiskB2;
   Slots: {FixedDiskES2}
end Computer&FixedDiskBoard CFDComp2.
```

The expansion of the fixed-disk storage works as described, with the expansion slots "FixedDiskES1" and "FixedDiskES2" signaling that the fixed-disk of "PC1" may be expanded, and the compatibility instances "CDDComp1" and "CDDComp2" telling which fixed-disks may be used for the expansion.

Diskette Drives

Drive position A of the IBM RT PC system unit always has a

IBM AT High Capacity Diskette Drive installed. Drive position B can have either the IBM AT High Capacity Diskette Drive or the IBM AT Dual-Sided Diskette Drive installed. These two drives are modelled in the ASL Catalogue by the objects "DisketteD1" and "DisketteD2", defined as follows:

```
Create DisketteDriver DisketteD1 with:
   Vendor: IBM;
   CompName: IBMATHighCapacityDisketteDrive;
   FormattedStorageCapability: 1200;
   AccessType: ReadWrite;
end DisketteDriver DisketteD1.

Create DisketteDriver DisketteD2 with:
   Vendor: IBM;
   CompName: IBMATDualSidedDisketteDrive;
   FormattedStorageCapability: 360;
   AccessType: ReadWrite;
end DisketteDriver DisketteD2.
```

The first object is included in the basic configuration of "PC1" by inserting it into the set which is the value of property "BuiltInRAStoragePeripherals" of "PC1". Property "BuiltInRAStorage" summarizes the total amount (in kilobytes) of secondary storage available in "PC1". The possibility of using both diskette drives as expansion boards for "PC1" is then modelled in the same way fixed-disk expansions have been modelled. First, the diskette expansion slot (representing drive position B) is created:

```
Create DisketteExpansionSlot DisketteES1.
```

Then, the drives are declared to be expansion boards by creating the appropriate diskette expansion board instances:

```
Create DisketteBoard DisketteB1 with:
   AllowedDisketteDriver: DisketteD1;
end DisketteBoard DisketteB1.

Create DisketteBoard DisketteB2 with:
   AllowedDisketteDriver: DisketteD2;
end DisketteBoard DisketteB2.
```

Finally, the compatibility of the expansion boards and the computer object is asserted:

```
Create Computer&DisketteBoard CDComp1 with:
   Computer: PC1;
   Board: DisketteB1;
   Slots: {DisketteES1};
end Computer&DisketteBoard CDComp1.

Create Computer&DisketteBoard CDComp2 with:
   Computer: PC1;
   Board: DisketteB2;
   Slots: {DisketteES1};
end Computer&DisketteBoard CDComp2.
```

7.1.6. External Interfaces

The external interfaces of the basic configuration of a computer are the built-in ports that can be used to connect the computer with external devices. In the basic configuration of the IBM RT PC, there are two built-in serial ports, both of type RS232, and the display, keyboard and mouse adapters, which are used to connect the computer system unit to the display, keyboard and mouse, respectively. This situation is modelled in the ASL Catalogue by defining the interface types representing these adapters; the display, keyboard and mouse adapter types have already been showed, they are objects "DADPT2in", "KADPT2in", and "MADPT2in", respectively; it remains to show the creation of the RS232 adapter:

```
Create SerialInterfaceType RS232 with:
   Syn/Asyn: A;
   Half/FullDuplex: HF;
   Input/Output: IO;
   MinimumRate: 50;
   MaximumRate: 19200;
end SerialInterfaceType RS232.
```

The objects representing the interfaces of our computer may now be created:

```
Create ExtInterface EI24 with:
   InterfaceType: RS232
end ExtInterface EI24.

Create ExtInterface EI25 with:
   InterfaceType: RS232;
end ExtInterface EI25.

Create ExtInterface EI26 with:
   InterfaceType: DADPT2in;
end ExtInterface EI26.

Create ExtInterface EI27 with:
   InterfaceType: KADPT2in;
end ExtInterface EI27.

Create ExtInterface EI28 with:
   InterfaceType: MADPT2in;
end ExtInterface EI28.
```

and inserted into the set which is the value of property "ExternalInterfaces" of object "PC1".

7.1.7. Expansion Slots

The IBM RT PC system unit has 7 generic expansion slots (or multi-use expansions slots, explained in Section 5.2.2.3), internally numbered from 2 to 8. Among them, slot 6 can only be used for the IBM PC Enhanced Graphics Adapter, by which the computer can be connected to the IBM 5154 Enhanced Color Display; slot 2, instead, may be alternatively used for 13 different adapters. The following operation creates all "PC1" generic expansion slot objects:

```
Create GenericExpansionSlot GenericES1.

Create GenericExpansionSlot GenericES2.

Create GenericExpansionSlot GenericES3.

Create GenericExpansionSlot GenericES4.

Create GenericExpansionSlot GenericES5.

Create GenericExpansionSlot GenericES6.

Create GenericExpansionSlot GenericES7.
```

Object "GenericES1" is intended to represent the generic slot 2 of the computer. Among the different adapters that can be installed in slot 2, there are serial ports, network ports and a graphics processor adapters. This means that the object "GenericES1" represents a slot that can be, among other things, an external, an interface, or a coprocessor expansion slot. This fact is represented by including object "GenericES1" also in classes "ExtInterfaceExpansionSlot, "NetInterfaceExpansionSlot, and "CoprocessorExpansionSlot, as follows:

```
Create ExtInterfaceExpansionSlot GenericES1.

Create NetInterfaceExpansionSlot GenericES1.

Create CoprocessorExpansionSlot GenericES1.
```

The other generic expansion slots are made instances of the appropriate classes in the same way. To show how a generic slot may be used in making an expansion to a computer, let us consider the following external interface expansion board, which, when installed in a computer, adds four RS422 serial ports to the computer:

```
Create ExtInterfaceBoard ExtIntB1 with:
   AllowedExtInterface: {EI34, EI35, EI36, EI37};
end ExtInterfaceBoard ExtIntB1.
```

where each object in the set given as value of property "AllowedExtInterface" is an external interface of type RS422. The compatibility instance that links this interface board to our computer is given by:

```
Create Computer&ExtInterfaceBoard CEIComp1 with:
   Computer: PC1;
   Board: ExtIntB1;
   Slots: {GenericES1, GenericES3, GenericES4, GenericES6,
           GenericES7};
   MaximumNumber: 4;
end Computer&ExtInterfaceBoard CEIComp1.
```

where the multivalued property "Slots" tells in which slots of "PC1" the board "ExtIntB1" may be installed, whereas property "MaximumNumber" gives the maximum number of boards "ExtIntB1" that can be placed in the computer. Of course, the number of slots in the "Slots" property value must be greater than or equal to the "MaximumNumber" value.

7.2. Software Definitions

The value of property "OperatingSystem" of a computer object gives the object representing the operating system running on the computer. For "PC1", we have the operating system defined as follows:

```
Create OperatingSystem UNIXSV with:
   Vendor: AT&T;
   CompName: Unix;
   CompVersion: SystemV;
   VirtualMemory: true;
   FileSystemType: Hierarchical;
   RequiredMainMemory: 512;
end OperatingSystem UNIXSV.
```

Analogously, the set value of property "AvailableLanguages" describes which are the programming language tools that come with the basic configuration of the computer. The C compiler for "PC1" is represented in the ASL Catalogue by the object created as follows:

```
Create ProgrammingLanguageTool CLang1 with:
   Vendor: AT&T;
   Language: C;
   ToolType: Compiler;
end ProgrammingLanguageTool CLang1.
```

The object "Fortran1", representing the FORTRAN compiler is defined in a similar way, as well as objects "Ed1" and "Emacs1", representing the one-line standard editor and

Emacs, respectively. From the software viewpoint, the most interesting property of a computer object is "AvailableOfficeSupportTools", whose value describes the software packages supporting office activities that are part of the basic configuration of the computer. The only free tool provided with "PC1", is the Ingres centralized database management system, defined by:

```
Create CentralizedDBMS Ingres1 with:
   Vendor: AT&T;
   CompName: Ingres;
   DataModel: Relational;
   NumberOfRecordsPerFile: 1000;
   RecordSize: 1024;
   FieldSize: 1024;
   RequiredMainMemory: 512;
end CentralizedDBMS Ingres1.
```

The "AvailableOfficeSupportTools" property is the one that most likely will be extended in developing the Architecture that supports the Office Information System being designed. Such extension is modelled as an expansion, namely a software expansion, with apposite compatibility classes describing the available options.

Other properties concerning software are "AvailableTerminalEmulators" and "AvailableNetSoftware"; both of them have a "nil" value to signify that the basic "PC1" configuration does not provide any terminal emulator or network software package.

7.3. Compatible Components

When inserting a new object in the ASL Catalogue, also the components which are point-to-point compatible with the object must be inserted, if not already in the Catalogue. If, on the other hand, the newly created component happens to be compatible with an already existing component, then only the compatibility between the two must be asserted. In this Section we show the insertion in the Catalogue of two components that are compatible with the IBM RT PC.

7.3.1. Displays

There are four displays that are point-to-point compatible with the IBM RT PC. We describe the creation of only one of them, namely the IBM 6154 Advanced Color Graphics Display. The insertion of the corresponding object is performed by the following operation:

```
Create Display Disp5 with:
   Vendor: IBM;
   CompName: IBM6154;
   ColorScreen: true;
   GraphicScreen: true;
   ExternalInterfaces: {EI31};
end Display Disp5.
```

where the external interface object "EI31" is created by:

```
Create ExtInterface EI31 with:
   InterfaceType: DADPT5out;
end ExtInterface EI31.
```

the type of "EI31" being a display adapter type previously defined. The point-to-point compatibility between the display and the computer is represented by the following object:

```
Create Computer&Display CDSComp3 with:
   Device1: PC1;
   Device2: Disp5;
   Interface1: DADPT5in;
   Interface2: DADPT5out;
end Computer&Display CDSComp3.
```

which asserts that computer "PC1" can be point-to-point connected to display "Disp5" through an external interface of type "DADPT5in", whereas an external interface of type "DADPT5out" must be used for the display. An interface of this type may not be (and in fact it is not) built-in "PC1", thus the knowledge base must be told that an external interface of type "DADPT5in" may be installed on "PC1" via an appropriate expansion operation. This is done by performing the following external interface expansion board creation:

```
Create ExtInterfaceBoard ExtIntB4 with:
   AllowedExtInterface: {EI40};
end ExtInterfaceBoard ExtIntB2.
```

where "EI40" is given by:

```
Create ExtInterfaceT EI40 with:
   InterfaceType: DADPT5in;
end ExtInterfaceT EI40.
```

and adapter "DADPT5in" is created in the way shown before for other adapters. The board "ExtIntB4" is linked to "PC1" via the compatibility instance given by:

```
Create Computer&ExtInterfaceBoard CEIComp4 with:
   Computer: PC1;
   Board: ExtIntB4;
   Slots: {GenericES1, GenericES3 ,GenericES4,
           GenericES6, GenericES7};
   MaximumNumber: 1;
end Computer&ExtInterfaceBoard CEIComp4.
```

As already explained, "CEIComp4" represents the fact that at most one "ExtIntB2" can be installed on "PC1", through one of the five generic slots that appear in the value of property "Slots".

7.3.2. Tape Drives

The IBM RT PC may have at most one cassette drive connected, the IBM 6157 Streaming Tape Drive, defined in the ASL Catalogue as follows:

```
Create CassetteDriver CassetteD1 with:
   Vendor: IBM;
   CompName: IBM6157;
   AccessType: ReadWrite;
   ExternalInterfaces: {EI33};
end CassetteDriver CassetteD1.
```

where the external interface object "EI33" is created by:

```
Create ExtInterface EI33 with:
   InterfaceType: TADPT1out;
end ExtInterface EI33.
```

the type of "EI33" being a display adapter type, given by:

```
Create ParallelInterfaceType TADPT1out with:
   InterfaceName: TADPT1;
```

```
    Input/Output: O;
  end ParallelInterfaceType TADPT1out.
```

Analogously to the previous case, the point-to-point compatibility between "PC1" and "CassetteD1" is declared by the object:

```
  Create Computer&CassetteDriver CCDComp1 with:
     Device1: PC1;
     Device2: CassetteD1;
     Interface1: TADPT1in;
     Interface2: TADPT1out;
  end Computer&CassetteDriver CCDComp1.
```

An adapter of the appropriate type is seen as an external interface board by creating the object:

```
  Create ExtInterfaceBoard ExtIntB6 with:
     AllowedExtInterface: {EI42};
  end ExtInterfaceBoard ExtIntB6.
```

which is related to "PC1" by:

```
  Create Computer&ExtInterfaceBoard CEIComp6 with:
     Computer: PC1;
     Board: ExtIntB6;
     Slots: {GenericES1, GenericES3 ,GenericES4,
             GenericES6, GenericES7};
     MaximumNumber: 1;
  end Computer&ExtInterfaceBoard CEIComp6.
```

REFERENCES

[Barb87] Barbic, F., Fugini, M.G., Maiocchi, R., Pernici, B., Rhames, J.R., and Rolland, C., 'C-TODOS: An Automatic Tool for Office System Conceptual Design', Politecnico di Milano, Electronics Dept., Rep. n. 87-15, 1987.

[Cast88] Castelli, D., Meghini, C., and Musto, D., 'Architecture Specification Language: Design and Implementation', TODOS Technical Report n. T4.2, in preparation.

[Bass87] Bassanini, G., Di Stefano, F., and Lunghi, G., 'TODOS Analysis Model Overview', TODOS Technical Report n. T1.2.2.1, July 1987.

[McDe80] McDermott, J., 'R1: A Rule-Based Configurer of Computer Systems', Technical Report n. CMU-CS-80-119, Carnegie-Mellon University, Dept. of Computer Science, 1980.

[Leve79] Levesque, H. and Mylopoulos, J., 'A Procedural Semantics for Semantic Networks', in 'Associative Networks', N. Findler (ed.), Academic Press, 1979.

[IBM] IBM RT PC Hardware Maintenance and Service Manual

[Pern86] Pernici, B. and Vogel, W., 'An Integrated Approach to OIS Development', ESPRIT Technical Week 86, Bruxelles, September 1986.

[Sowa84] Sowa, J. F., 'Conceptual Structures', Addison-Wesley, 1984.

[Stal84] Stalling, W., 'Local Networks', ACM Computing Surveys, 16 (1), March 1984.

[Wood75] Woods, W., 'What's in a Link: Foundations for Semantic Networks', in 'Representation and Understanding', D.G. Bobrow and A.M. Collins (eds.), New York, Academic Press, 1975.

Appendix A: The ASL Is-a Hierarchy

As a notational convention, the ASL is-a hierarchy is illustrated by representing specialization by indentation, so that the more general classes are the less indented. We have divided the is-a hierarchy as follows: the hierarchy of ASL metaclasses, that of the Catalogue classes, that of the Compatibility classes, and finally that of Architectural Units classes.

A.1. Metaclasses Is-a Hierarchy

```
ASLClass
   CatalogueClass
      CompatibilityClass
      CatalogueItemClass
   ArchitecturalUnitClass
```

A.2. Catalogue Is-a Hierarchy

```
Catalogue
   Component
      SoftwareComponent
         SoftwareTool
            OfficeSupportTool
               VoiceProcessingTool
               ImageProcessingTool
                  GraphicTool
               DataProcessingTool
                  Calendar
                  Scheduling
                  SpreadSheet
                  WordProcessingTool
                  DBMS
                     DistributedDBMS
                     CentralizedDBMS
            SystemSoftware
               NetSystemSoftware
               ElectronicMail
               Editor
               ProgrammingLanguageTool
         OperatingSystem
```

```
HardwareComponent
   ExpandableComponent
      Computer
         NetworkServer
         WordProcessor
         IntegratedWorkstation
         PersonalComputer
         MiniComputer
         MainFrame
      Peripheral
         StoragePeripheral
            ExternalStoragePeripheral
               ExternalRAStoragePeripheral
                  ExternalDisketteDriver
                  ExternalOpticalDiskDriver
                  ExternalMagneticDiskDriver
               ExternalSAStoragePeripheral
                  CartridgeDriver
                  CassetteDriver
                  TapeDriver
            InternalStoragePeripheral
               InternalSAStoragePeripheral
                  InternalCassetteDriver
                  InternalTapeDriver
                  InternalCartridgeDriver
               InternalRAStoragePeripheral
                  InternalDisketteDriver
                  InternalOpticalDiskDriver
                  InternalMagneticDiskDriver
         IOPeripheral
            Terminal
               VideoTerminal
               PrinterTerminal
            Printer
            Plotter
            Scanner
            Display
            Keyboard
            PointingDevice
   StaticComponent
      LocalAreaNet
         TreeLocalAreaNet
         RingLocalAreaNet
         BusLocalAreaNet
      ExpansionBoard
         NetInterfaceBoard
         ExtInterfaceBoard
         StorageBoard
```

```
            CartridgeBoard
            DisketteBoard
            FixedDiskBoard
         MainMemoryBoard
         CoprocessorBoard
      CPU
      ExpansionSlot
         GenericExpansionSlot
         NetInterfaceExpansionSlot
         ExtInterfaceExpansionSlot
         CoprocessorExpansionSlot
         StorageExpansionSlot
            CartridgeExpansionSlot
            FixedDiskExpansionSlot
            DisketteExpansionSlot
      Interface
         NetInterface
         ExtInterface
AuxCatalogue
   AccessType
   PrinterType
   Font
   DataForm
   ScreenDimension
   ScreenType
   PointingDeviceType
   TransmissionTechnique
   LanProtocol
      RingLanProtocol
      BusTreeLanProtocol
   TransmissionMedium
   CoprocessorFunctionality
   PowerRequirements
   MechanicalDimension
   Environment
   ExtInterfaceRate
   InputOutput
   HalfDuplexFullDuplex
   SynchAsynch
   InterfaceType
      NetInterfaceType
      ExtInterfaceType
         BusInterfaceType
         ParallelInterfaceType
         SerialInterfaceType
   RepresentationForm
   InteractionMode
   DataModel
```

GraphicalObject
CursorPositioner
Language
ToolType
FileSystem
Pnumber
Vendor
Boolean

A.3. Compatibility Is-a Hierarchy

```
Compatibility
   ExpansionCompatibility
      SoftwareExpansionCompatibility
      HardwareExpansionCompatibility
         Computer&CartridgeBoard
         Computer&NetInterfaceBoard
         Computer&DisketteBoard
         Computer&FixedDiskBoard
         Computer&ExtInterfaceBoard
         Computer&CoprocessorBoard
         Computer&MainMemoryBoard
   PTPCompatibility
      MagneticDiskDriver&MagneticDiskDriver
      Computer&Terminal
      Computer&Scanner
      Computer&Printer
      Computer&Plotter
      Computer&Display
      Computer&MagneticDiskDriver
      Computer&OpticalDiskDriver
      Computer&DisketteDriver
      Computer&TapeDriver
      Computer&CassetteDriver
      Computer&CartridgeDriver
      Computer&Computer
```

A.4. Architectural Units Is-a Hierarchy

```
ArchitecturalUnit
   Architecture
      ComplexArchitecture
      Subsystem
         PTPConnectedSubsystem
         ComputerUnit
            MainFrameUnit
            MiniComputerUnit
            PersonalComputerUnit
            IntegratedWorkstationUnit
            WordProcessorUnit
            NetworkServerUnit
   OfficeNetwork
      TreeNetwork
      RingNetwork
      BusNetwork
   Subsystem
      PTPConnectedSubsystem
      ComputerUnit
         MainFrameUnit
         MiniComputerUnit
         PersonalComputerUnit
         IntegratedWorkstationUnit
         WordProcessorUnit
         NetworkServerUnit
   Unit
      SoftwarePackage
         SoftwareToolUnit
            SystemSoftwareUnit
               ProgrammingLanguageToolUnit
               EditorUnit
               ElectronicMailUnit
               NetSystemSoftwareUnit
            OfficeSupportToolUnit
               DataProcessingToolUnit
                  DBMSUnit
                     CentralizedDBMSUnit
                     DistributedDBMSUnit
                  WordProcessingToolUnit
                  SpreadSheetUnit
                  SchedulingUnit
                  CalendarUnit
               ImageProcessingToolUnit
                  GraphicToolUnit
               VoiceProcessingToolUnit
         OperatingSystemUnit
```

```
HardwareUnit
   ExpandableUnit
      PeripheralUnit
         StoragePeripheralUnit
            ExternalStoragePeripheralUnit
               ExternalSASPUnit
                  CartridgeUnit
                  CassetteUnit
                  TapeUnit
               ExternalRASPUnit
                  ExternalDisketteUnit
                  ExternalOpticalDiskUnit
                  ExternalMagneticDiskUnit
            InternalStoragePeripheralUnit
               InternalSASPUnit
                  InternalCartridgeUnit
                  InternalCassetteUnit
                  InternalTapeUnit
               InternalRASPUnit
                  InternalDisketteUnit
                  InternalOpticalDiskUnit
                  InternalMagneticDiskUnit
         IOPeripheralUnit
            TerminalUnit
               PrinterTerminalUnit
               VideoTerminalUnit
            PrinterUnit
            PlotterUnit
            ScannerUnit
            DisplayUnit
            KeyboardUnit
            PointingDeviceUnit
      ComputerUnit
         MainFrameUnit
         MiniComputerUnit
         PersonalComputerUnit
         IntegratedWorkstationUnit
         WordProcessorUnit
         NetworkServerUnit
   StaticUnit
      ExpansionSlotUnit
         StorageExpansionSlotUnit
            DisketteExpansionSlotUnit
            FixedDiskExpansionSlotUnit
            CartridgeExpansionSlotUnit
         CoprocessorExpansionSlotUnit
         ExtInterfaceExpansionSlotUnit
         NetInterfaceExpansionSlotUnit
```

```
            GenericExpansionSlotUnit
         CPUUnit
         ExpansionBoardUnit
            ExtInterfaceBoardUnit
            NetInterfaceBoardUnit
            StorageBoardUnit
               DisketteBoardUnit
               CartridgeBoardUnit
               FixedDiskBoardUnit
            MainMemoryBoardUnit
            CoprocessorBoardUnit
         LocalAreaNetUnit
            BusLANUnit
            RingLANUnit
            TreeLANUnit
         InterfaceUnit
            NetInterfaceUnit
            ExtInterfaceUnit
Expansion
   SoftwareExpansion
   HardwareExpansion
      DisketteExpansion
      FixedDiskExpansion
      CartridgeExpansion
      NetInterfaceExpansion
      ExtInterfaceExpansion
      CoprocessorExpansion
      MainMemoryExpansion
AuxArchitecturalUnit
   Gateway
   SubsystemPair
   Host
   PTPConnection
```

Appendix B: The ASL Interface Operations

This Appendix lists the interface operations, following the same order of Section 4. For each operation, inputs, returned value, and the conditions that cause it to fail, are given. In addition, a brief description of the operation behavior is presented. Not all this functions have been implemented in the ASL prototype. However, we present the complete list in order to give an full account of the ASL language.

This Section is structured as follows: in the first Section we will describe the operations available in the Catalogue Administrator Interface; in the second Section the operations of the ASL Interface are presented, collected by level.

The following notational conventions are used:

{ l } stands for the list l

a | b | c stands for the alternatives a,b,c

B.1. Catalogue Administrator Interface Operations

is-a

Input	nil
Return	nil
Failures	nil
Description	The is-a hierarchy is displayed.

get_component

Input	C: CatalogueClass, CD: Condition
Return	List of C objects
Failures	C is not a class of the Catalogue. An intermediate condition selection does not result in a single value.
Description	It selects the objects of the class C that satisfy the condition

display_class_definition

Input	C: CatalogueClass
Return	nil
Failures	C is not a Catalogue Class.
Description	A list of the properties of class C. For each property, the name, type, and default value, if any, is displayed.

display_component

Input	O: Catalogue
Return	nil
Failures	O does not belong to anyone of the Catalogue classes.
Description	The list of the values of the slots and relations of O is displayed.

instance_of

Input	O: Catalogue
Return	List of Catalogue classes
Failures	O does not belong to anyone of the Catalogue classes.
Description	The list of the Catalogue classes of which O is an instance, is returned.

part_of

Input	O: Catalogue
Return	List of <DomainComponent, Slot|Relation> pairs.
Failures	O does not belong to anyone of the Catalogue classes.
Description	A list of <DomainComponent Slot|Relation> pairs for each Slot or Relation having O either as range value or as member of the range value (for relations or set-valued properties), is returned.

create_component

Input	C: CatalogueItemClass, N: atom L: List of {Slot|Relation Value}
Return	The newly created object N.
Failures	N is the name of an already existing object. C is not a class of the Catalogue. Value does not belong to the Slot|Relation range class.
Description	A new instance is added to the class C of the Catalogue.

delete_component

Input	O: Catalogue
Return	nil
Failures	O is part of some existing Knowledge Base object.
Description	The input object is removed from the Catalogue.

create_version

Input	O: Catalogue, N: atom L: List of {Slot|Relation Value},
Return	Catalogue object
Failures	O is not an object of the Catalogue. N is the name of an already existing object.
Description	An object is added to the Catalogue. The new object differs from O in the value of the properties in L.

delete_versions

Input	O: Catalogue,
Return	nil
Failures	O is not an object of the Catalogue. Some of the configurations to be deleted are part of existing objects.
Description	All the versions of O are removed from the Catalogue.

get_versions

Input O: Catalogue
Return List of Catalogue objects
Failures O is not an object of the Catalogue

Description All the versions of object O are returned.

create_main_memory_board_compatibility

Input O1: Computer,
O2: MainMemoryBoard,
N: atom
Return Computer&MainMemoryBoard object
Failures N is the name of an already existing object.

Description A new object is created that describes the compatibility between the computer O1 and the main memory board O2.

create_coprocessor_board_compatibility

Input O1: Computer,
O2: CoprocessorBoard,
O3: {CoprocessorExpansionSlot},
X: number,
N: atom
Return Computer&CoprocessorBoard object
Failures N is the name of an already existing object.

Description A new object is created that describes the compatibility between the computer O1 and the coprocessor board O2.
O3 is the list of O1 slots that are compatible with the O2 board. X is the maximum number of O2 boards allowed for the computer O1.

create_external_interface_board_compatibility

Input	O1: Computer, O2: ExtInterfaceBoard, O3: {ExtInterfaceExpansionSlot}, X: number, N: atom
Return	Computer&ExtInterfaceBoard object
Failures	N is the name of an already existing object.
Description	A new object is created that describes the compatibility between the computer O1 and the external interface board O2. O3 is the list of O1 slots that are compatible with the O2 board. X is the maximum number of O2 boards allowed for the computer O1.

create_fixed_disk_board_compatibility

Input	O1: Computer, O2: FixedDiskBoard, O3: {FixedDiskExpansionSlot}, X: number, N: atom
Return	ComputerFixedDiskBoard object
Failures	N is the name of an already existing object
Description	A new object is created that describes the compatibility between the computer O1 and the fixed disk board O2. O3 is the list of O1 slots that are compatible with the O2 board. X is the maximum number of O2 boards allowed for the computer O1.

create_diskette_board_compatibility

Input	O1: Computer, O2: DisketteBoard, O3: {DisketteExpansionSlot}, X: number, N: atom
Return	Computer&DisketteBoard object
Failures	N is the name of an already existing object.
Description	A new object is created that describes the compatibility between the computer O1 and the diskette expansion board O2. O3 is the list of O1 slots that are compatible with the O2 board. X is the maximum number of O2 boards allowed for the computer O1.

create_net_interface_board_compatibility

Input	O1: Computer, O2: NetInterfaceBoard, O3: {NetInterfaceExpansionSlot}, X: number, N: atom
Return	Computer&NetInterfaceBoard object
Failures	N is the name of an already existing object.
Description	A new object is created that describes the compatibility between the computer O1 and the network interface board O2. O3 is the list of O1 slots that are compatible with the O2 board. X is the maximum number of O2 boards allowed for the computer O1.

create_cartridge_board_compatibility

Input	O1: Computer, O2: CartridgeBoard, O3: {CartridgeExpansionSlot}, X: number, N: atom
Return	Computer&CartridgeBoard object
Failures	N is the name of an already existing object.
Description	A new object is created that describes the compatibility between the computer O1 and the cartridge board O2. O3 is the list of O1 slots that are compatible with the O2 board. X is the maximum number of O2 boards allowed for the computer O1.

create_software_compatibility

Input	O1: Computer, O2: SoftwareTool, O3: {SoftwareTool}, N: atom
Return	SoftwareCompatibility object
Failures	N is the name of an already existing object.
Description	A new object is created that describes the compatibility between the software tool O2 and the computer O1. O3 is the software required to run O2 on O1.

create_computer_computer_compatibility

Input	O1: Computer, O2: Computer, O3: InterfaceType, O4: InterfaceType, N: atom
Return	Computer&Computer object
Failures	N is the name of an already existing object.
Description	A new Point-to-Point compatibility description is created between O1 and O2.

create_computer_cartridge_driver_compatibility

Input	O1: Computer, O2: CartridgeDriver, O3: InterfaceType, O4: InterfaceType, N: atom
Return	Computer&CartridgeDriver object
Failures	N is the name of an already existing object.
Description	A new Point-to-Point compatibility description is created. O3 and O4 are the required types of external interfaces for O1 and O2 respectively.

create_computer_cassette_driver_compatibility

Input	O1: Computer, O2: CassetteDriver, O3: InterfaceType, O4: InterfaceType, N: atom
Return	Computer&CassetteDriver object
Failures	N is the name of an already existing object.
Description	A new Point-to-Point compatibility description is created. O3 and O4 are the required types of external interfaces for O1 and O2 respectively.

create_computer_tape_driver_compatibility

Input	O1: Computer, O2: TapeDriver, O3: InterfaceType, O4: InterfaceType, N: atom
Return	Computer&TapeDriver object
Failures	N is the name of an already existing object.
Description	A new Point-to-Point compatibility description is created. O3 and O4 are the required types of external interfaces for O1 and O2 respectively.

create_computer_diskette_driver_compatibility

Input	O1: Computer, O2: DisketteDriver, O3: InterfaceType, O4: InterfaceType , N: atom
Return	Computer&DisketteDriver object
Failures	N is the name of an already existing object.
Description	A new Point-to-Point compatibility description is created. O3 and O4 are the required types of external interfaces for O1 and O2 respectively.

create_computer_optical_disk_driver_compatibility

Input	O1: Computer, O2: OpticalDiskDriver, O3: InterfaceType, O4: InterfaceType, N: atom
Return	Computer&OpticalDiskDriver object
Failures	N is the name of an already existing object.
Description	A new Point-to-Point compatibility description is created. O3 and O4 are the required types of external interfaces for O1 and O2 respectively.

create_computer_magnetic_disk_driver_compatibility

Input	O1: Computer, O2: MagneticDiskDriver, O3: InterfaceType, O4: InterfaceType , N: atom
Return	Computer&MagneticDiskDriver object
Failures	N is the name of an already existing object.
Description	A new Point-to-Point compatibility description is created. O3 and O4 are the required types of external interfaces for O1 and O2 respectively.

create_computer_display_compatibility

Input	O1: Computer, O2: Display, O3: InterfaceType, O4: InterfaceType, N: atom
Return	Computer&Display object
Failures	N is the name of an already existing object.
Description	A new Point-to-Point compatibility description is created. O3 and O4 are the required types of external interfaces for O1 and O2 respectively.

create_computer_plotter_compatibility

Input	O1: Computer, O2: PlotterDisplay, O3: InterfaceType, O4: InterfaceType, N: atom
Return	Computer&Plotter object
Failures	N is the name of an already existing object.
Description	A new Point-to-Point compatibility description is created. O3 and O4 are the required types of external interfaces for O1 and O2 respectively.

create_computer_scanner_compatibility

Input	O1: Computer, O2: Scanner, O3: InterfaceType, O4: InterfaceType, N: atom
Return	Computer&Scanner object
Failures	N is the name of an already existing object.
Description	A new Point-to-Point compatibility description is created. O3 and O4 are the required types of external interfaces for O1 and O2 respectively.

create_magnetic_disk_driver_magnetic_disk_driver_compatibility

Input	O1: MagneticDiskDriver, O2: MagneticDiskDriver, O3: InterfaceType, O4: InterfaceType, N: atom
Return	MagneticDiskDriver&MagneticDiskDriver object
Failures	N is the name of an already existing object.
Description	A new Point-to-Point compatibility description is created. O3 and O4 are the required types of external interfaces for O1 and O2 respectively.

delete_main_memory_board_compatibility

Input	O: Computer&MainMemoryBoard
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_coprocessor_board_compatibility

Input	O: Computer&CoprocessorBoard
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_external_interface_board_compatibility

Input	O: Computer&ExtInterfaceBoard
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_fixed_disk_board_compatibility

Input	O: ComputerFixedDiskBoard
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_diskette_board_compatibility

Input	O: Computer&DisketteBoard
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_net_interface_board_compatibility

Input	O: Computer&NetInterfaceBoard
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_cartridge_board_compatibility

Input	O: Computer&CartridgeBoard
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_software_compatibility

Input	O: SoftwareCompatibility
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_computer_computer_compatibility

Input	O: Computer&Computer
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_computer_cartridge_driver_compatibility

Input	O: Computer&CartridgeDrive
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_computer_cassette_driver_compatibility

Input	O: Computer&CassetteDriver
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from th Catalogue.

delete_computer_tape_driver_compatibility

Input	O: Computer&TapeDriver
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_computer_diskette_driver_compatibility

Input	O: Computer&DisketteDriver
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_computer_optical_disk_driver_compatibility

Input	O: Computer&OpticalDiskDriver
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_computer_magnetic_disk_driver_compatibility

Input	O: Computer&MagneticDiskDriver
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_computer_display_compatibility

Input	O: Computer&Display
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_computer_plotter_compatibility

Input	O: Computer&Plotter
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_computer_scanner_compatibility

Input	O: Computer&Scanner
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

delete_magnetic_disk_driver_magnetic_disk_driver_compatibility

Input	O: MagneticDiskDriver&MagneticDiskDriver
Return	nil
Failures	O is not a compatibility object
Description	The compatibility object O is deleted from the Catalogue.

B.2. The ASL Interface Operations

In this Section, the operations on architectural Units will be detailed, group by level of abstraction.

B.2.1. Catalogue Level Operations

The operations of this level are the subset of the Catalogue Administrator Interface operations consisting of the operations that do not modify the ASL Catalogue. In particular, these functions are: is-a, get_component, display_class_definition, display_component, instance_of, part_of, get_versions. Their description is not repeated.

B.2.2. Unit Level Operations

get_unit

Input	C: UnitClass, CD: Condition
Return	List of C objects that satisfy the condition.
Failures	None
Description	Selects the objects of the class C that satisfy the given condition.

display_unit

Input	O: Unit
Return	nil
Failures	O is not an existing Unit.
Description	The list of the object slot and relation values is displayed.

instance_of

Input	O: Unit
Return	List of Unit level classes
Failures	O does not belong to anyone of the Unit Level classes.
Description	The list of Unit level classes of which O is an instance is returned.

part_of

Input	O: Unit
Return	List of <DomainUnit, Slot|Relation> pairs.
Failures	O does not belong to anyone of the Unit classes.
Description	A list of <DomainUnit, Slot|Relation> pairs for each Slot or Relation having O either as range value or as member of the range value (for relations or set-valued properties), is returned.

create_unit

Input	N: atom, O: Component
Return	Unit object.
Failures	N is the name of an already existing object.
Description	A Unit object and its component Unit objects are created.

delete_unit

Input	O: Unit
Return	nil
Failures	O is not an existing Unit. O is a component of some PTP connected Subsystem.
Description	O and its component Units are deleted.

expand_with_main_memory_board

Input	O: ComputerUnit, X: number,
Return	ComputerUnit object.
Failures	O has no free expansion slot. O has no compatible memory boards. The resulting installed main memory exceeds the maximum main memory of O basic model.
Description	O is modified by adding a new memory expansion board which brings the total main memory of O to X.

expand_with_coprocessor_board

Input	O1: ComputerUnit, O2: CoprocessorFunctionality.
Return	ComputerUnit object.
Failures	O1 has no free expansion slot. O1 has no compatible coprocessor boards.
Description	O1 is modified by adding a coprocessor expansion board Unit having the functionality O2.

expand_with_external_interface_board

Input	O1: ComputerUnit, O2: ExtInterfaceType.
Return	ComputerUnit object.
Failures	O1 has no free expansion slot. O2 is not compatible with any free expansion slot of O1.
Description	O1 is modified by adding the external interface board Unit whose basic model is O2.

expand_with_network_interface_board

Input	O1: ComputerUnit, O2: NetInterfaceType.
Return	ComputerUnit object.
Failures	O1 has no free expansion slot. O2 is not compatible with any free expansion slot of O1.
Description	O1 is modified by adding the network interface board Unit whose basic model is O2.

expand_with_fixed_disk_board

Input	O1: ComputerUnit, O2: FixedDisk.
Return	ComputerUnit object.
Failures	O1 has no free expansion slot. O2 is not compatible with any free expansion slot of O1.
Description	O1 is modified by adding the fixed disk board Unit whose basic model is O2.

expand_with_diskette_board

Input	O1: ComputerUnit, O2: Diskette.
Return	ComputerUnit object.
Failures	O1 has no free expansion slot. O2 is not compatible with any free expansion slot of O1.
Description	O1 is modified by adding the diskette board Unit whose basic model is O2.

expand_with_cartridge_board

Input	O1: ComputerUnit, O2: Cartridge.
Return	ComputerUnit object.
Failures	O1 has no free expansion slot. O2 is not compatible with any free expansion slot of O1.
Description	O1 is modified by adding the cartridge board Unit whose basic model is O2.

expand_with_software

Input	O1: ComputerUnit, O2: SoftwarePackage.
Return	ComputerUnit object.
Failures	O1 has not enough memory to run O2. O1 has not the required software to run O2.
Description	The software package O2 is added to the set of software expansions of O1.

remove_main_memory_board

Input	O: ComputerUnit, X: number.
Return	ComputerUnit object.
Failures	O memory expansion is less than X.
Description	The memory expansion board Unit of O is replaced with a new expansion board Unit which brings to X the total main memory of O. If such dimension is 0 (or less) no expansion board Unit is created. The old expansion Unit is deleted.

remove_coprocessor_board

Input	O1: ComputerUnit, O2: CoprocessorFunctionality.
Return	ComputerUnit object.
Failures	O1 does not have any coprocessor expansion board Unit with the functionality O2.
Description	A coprocessor expansion board Unit with the functionality O2 is removed from the set added coprocessor boards and hardware expansion of O1. The old expansion board Unit is deleted.

remove_external_interface_board

Input	O1: ComputerUnit, O2: ExtInterfaceType
Return	ComputerUnit object.
Failures	O2 is not an external interface board of O1. There is an external interface of O2 that is not free.
Description	The set of external interfaces of O2 is removed from the set of free external interfaces of O1. O2 is removed from the set of hardware expansions of O1.

remove_network_interface_board

Input	O1: ComputerUnit, O2: NetInterfaceType.
Return	ComputerUnit object.
Failures	O2 is not a network interface board of O1. There is a network interface of O2 that is not free.
Description	The set of network interfaces of O2 is removed from the set of free network interfaces of O1. O2 is removed from the set of hardware expansions of O1.

remove_fixed_disk_board

Input	O1: ComputerUnit, O2: FixedDisk.
Return	ComputerUnit object.
Failures	O2 is not an expansion board of O1.
Description	O2 is removed from the set of hardware expansions of O1.

remove_diskette_board

Input	O1: ComputerUnit, O2: Diskette.
Return	ComputerUnit object.
Failures	O2 is not an expansion board of O1.
Description	O2 is removed from the set of hardware expansions of O1.

remove_cartridge_board

Input	O1: ComputerUnit, O2: Cartridge.
Return	ComputerUnit object.
Failures	O2 is not an expansion board of O1.
Description	O2 is removed from the set of hardware expansions of O1.

remove_software_expansion

Input	O1: ComputerUnit, O2: SoftwarePackage.
Return	ComputerUnit object.
Failures	O2 is not a software expansion of O1.
Description	The software package O2 is removed from the set of software expansion of O1.

B.2.3. Subsystem Level Operations

get_subsystem

Input	C: SubsystemClass, CD: Condition
Return	List of C objects that satisfy the condition.
Failures	none
Description	Selects the Subsystem objects that satisfy the condition.

display_subsystem

Input	O: PTPConnectedSubsystem
Return	nil
Failures	O is not an existing PTP connected Subsystem.
Description	The list of the Units and of the PTP connections (Unit pairs) of O is displayed.

create_subsystem

Input	N: atom, S1: {ExpandableUnit}, S2: {{O1: ExpandableUnit, O2: ExpandableUnit}}
Return	PTPConnectedSubsystem object.
Failures	S1 is empty. S2 is empty. N is the name of an already existing Subsystem. An element of S1 already belongs to another Subsystem. There is a Unit in S1 that has been specified more than once. There is a pair in S2 that has been specified more than once, either in the same or in reverse order. An element of some pair of S2 is not in S1. S1 does not contain any computer. The graph representing the resulting Subsystem is not serially connected.
Description	A PTP connected Subsystem object is created having the elements of S1 as component Units and the pairs in S2 as specification of the PTP links.

delete_subsystem

Input	O: Subsystem
Return	nil
Failures	O is part of some existing Network.
Description	O is deleted.

add_unit_to_subsystem

Input	O1: PTPConnectedSubsystem, O2: ExpandableUnit, S1: {{O3: ExpandableUnit, O4: ExpandableUnit}}.
Return	PTPConnectedSubsystem object.
Failures	O2 already belongs to some Subsystem. An element of some pair of S1 is neither an element of O1 or is equal to O2. The graph representing the resulting Subsystem is not serially connected. There is more than one link between some pair of the resulting Subsystem components.
Description	The expandable Unit O2 is added to the Subsystem O1 by establishing the links described in S1.

add_ptp_connection_to_subsystem

Input	O1: PTPConnectedSubsystem, P1: {O2: ExpandableUnit, O3: ExpandableUnit}.
Return	PTPConnectedSubsystem object.
Failures	Either O2 or O3 are not elements of O1. There is more than one link between some pair of the resulting Subsystem components.
Description	O1 is modified by adding the new PTP connection given by P1.

remove_unit_from_subsystem

Input	O1: PTPConnectedSubsystem, O2: ExpandableUnit.
Return	PTPConnectedSubsystem object.
Failures	O2 is not a Unit of O1. O2 is the only Unit of O1. The resulting Subsystem would not be serially connected.
Description	The expandable Unit O2 and the links that refer it are removed from the set of component Units of O1.

remove_ptp_connection_from_subsystem

Input: O1: PTPConnectedSubsystem,
P1: {O1: ExpandableUnit,
O2: ExpandableUnit}.
Return: PTPConnectedSubsystem object
Failures: The specified link does not belong to O1.
The graph representing the resulting Subsystem is not serially connected.

Description: The PTP connection object given by P1 is removed from the set of PTP connections of Subsystem O1.

in_subsystem

Input: O: ExpandableUnit
Return: Subsystem object
Failures: none

Description: It returns the Subsystem where O belongs.

ptp_connected_subsystem

Input: O: Subsystem
Return: Boolean value
Failures: none

Description: It returns true if O is an instance of class PTPConnectedSubsystem, nil otherwise.

B.2.4. Network Level Operations

get_network

Input: C: NetworkClass,
CD: Condition
Return: List of C objects that satisfy the condition.
Failures: none

Description: Selects the Networksthat satisfy the condition.

display_network

Input	O: Network
Return	nil
Failures	O is not an existing Network.
Description	The Network communication device and the list of the component Subsystems are displayed.

create_bus_network

Input	N: atom, O1: LocalAreaNetwork object, S1: {{O2: Subsystem, O3: ComputerUnit}}, S2: {O4: Subsystem}
Return	BusNetwork object
Failures	O3 is not a component Unit of O2. O4 does not appear as O2 element in S1. O3 does not have any free net interface compatible with O1.
Description	A bus Network is created whose communication device is O1 and whose component Subsystems are the O2 elements of S1. O3 is the computer Unit through which the Subsystem O2 is physically connected to the Network. For such reason O3 is modified making used one of its free net interface. The Subsystems are placed along the communication device according the order specified in S2.

create_ring_network

Input	N: atom, O1: LocalAreaNetwork object, S1: {{O2: Subsystem, O3: ComputerUnit}} S2: {O4: Subsystem}
Return	RingNetwork object
Failures	O3 is not a component Unit of O2. O4 does not appear as O2 element in S1. O3 does not have any free net interface compatible with O1.
Description	A ring Network is created whose communication device is O1 and whose component Subsystems are the O2 elements in S1. O3 is the Unit through which the Subsystem O2 is physically connected to the Network. For such reason O3 is modified making used one of its free net interface.

create_tree_network

Input	N: atom, O1: LocalAreaNetwork object, S1: {{O2: Subsystem, O3: ComputerUnit}} S2: {O4: Subsystem},
Return	TreeNetwork object
Failures	O3 is not a component Unit of O2. O4 do not appear as O2 element in S1. O3 does not have any free net interface compatible with O1.
Description	A tree Network is created whose communication device is O1 and whose component Subsystems are the O2 elements of S1. O3 is the Unit through which the Subsystem O2 is physically connected to the Network For such reason O3 is modified making used one of its free net interface. The elements in S2 describe the Network topology, each pair describing the relationship father-child between two Subsystems. The order among the pairs correspond to the order among the children in the same family.

add_subsystem_to_bus_network

Input	O1: BusNetwork, P1: {O2: Subsystem, O3: ComputerUnit}, O4: Subsystem.
Return	BusNetwork object
Failures	O3 is not a component Unit of O4. O4 is not a O1 component Subsystem. O3 does not have any free net interface compatible with O1.
Description	The Subsystem O2 is connected to the Network O1 through the O2 Unit O3. The new Subsystem is located as immediate successor of the Subsystem O4. O3 is modified making used one of its free network interface Units.

add_subsystem_to_ring_network

Input	O1: RingNetwork, P1: {O2: Subsystem, O3: ComputerUnit}, O4: Subsystem.
Return	RingNetwork object
Failures	O3 is not a component Unit of O4. O4 does not appear as O2 element in S1. O3 does not have any free net interface compatible with O1.
Description	The Subsystem O2 is connected to the Network O1 through the O2 Unit O3. The new Subsystem is located as immediate successor of the Subsystem O4. O3 is modified making used one of its free net interface.

add_subsystem_to_tree_network

Input	O1: TreeNetwork, P1: {O2: Subsystem, O3: ComputerUnit}, O4: Subsystem, O5: Subsystem.
Return	TreeNetwork object
Failures	O3 is not a component Unit of O4. O4 is not a O1 component Subsystem. O3 does not have any free net interface compatible with O1.
Description	The Subsystem O2 is connected to the Network O1 through the O2 Unit O3. The new Subsystem is located in the family of the Subsystem O4. O5 is the child of O4 that immediate precedes O2. O3 is modified making used one of its free net interface.

remove_subsystem_from_bus_network

Input	O1: BusNetwork object, O2: Subsystem.
Return	BusNetwork object
Failures	O2 is not a component Subsystem of the Network O1. O2 is the only Subsystem component of the Network O1.
Description	The Subsystem O2 is removed from Network O1. The computer Unit through which O2 is connected to O1 is modified making free the Network interface that connects O1 and O2.

remove_subsystem_from_ring_network

Input	O1: RingNetwork object, O2: Subsystem.
Return	RingNetwork object
Failures	O2 is not a component Subsystem of the Network O1. O2 is the only Subsystem component of the Network O1.
Description	The Subsystem O2 is removed from the Network O1. The computer Unit through which O2 is connected to O1 is modified making free the Network interface that connects O1 and O2.

remove_subsystem_from_tree_network

Input	O1: TreeNetwork object, O2: Subsystem.
Return	TreeNetwork object
Failures	O2 is not a component Subsystem of the Network O1. O2 is the only Subsystem component of the Network O1. The family of O2 must be empty.
Description	The Subsystem O2 is removed from Network O1. The computer Unit through which O2 is connected to O1 is modified making free the Network interface that connects O1 and O2.

delete_ring_network

Input	O1: RingNetwork
Return	nil
Failures	O1 is part of some existing Architecture.
Description	The ring Network O1 is deleted. The computer Units through which the O1 component Subsystems are connected to the Network are modified making free the Network interface that connects the Subsystem to the Network.

delete_bus_network

Input	O1: BusNetwork
Return	nil
Failures	O1 is part of some existing Architecture.
Description	The bus Network O1 is deleted. The computer Units through which the O1 component Subsystems are connected to the Network are modified making free the Network interface that connects the Subsystem to the Network.

delete_tree_network

Input	O1: TreeNetwork
Return	nil
Failures	O1 is part of some existing Architecture.
Description	The tree Network O1 is deleted. The computer Units through which the O1 component Subsystems are connected to the Network are modified making free the Network interface that connects the Subsystem to the Network.

B.2.5. Architecture Level Operations

get_architecture

Input	C: ArchitectureClass, CD: Condition
Return	List of complex Architectures that satisfy the specified condition.
Failures	none
Description	Selects the objects of the class C that satisfy the condition.

display_architecture

Input	O: ComplexArchitecture
Return	nil
Failures	O is not an existing Architecture.
Description	The set of Networks that belong to the Architecture and a description of their gates are displayed.

create_architecture

Input	N: atom, S1: {Network}
Return	ComplexArchitecture object
Failures	The resulting Architecture must be serially connected. Some of the the Networks in S1 are already part of some existing Architecture.
Description	A new Architecture is created. The gates between two component Networks are those Subsystems that are shared by the Networks.

delete_architecture

Input	O: ComplexArchitecture.
Return	nil
Failures	none
Description	The Architecture object O is deleted.

add_network_to_architecture

Input	O1: ComplexArchitecture, O2: Network object
Return	Architecture object
Failures	O2 already belongs to an Architecture. None of the O2 Subsystems belongs to any of O1 Networks (no gate between O1 and O2).
Description	O2 is added to the O1 component Networks.

remove_network_from_architecture

Input	O1: ComplexArchitecture, O2: Network
Return	Architecture object
Failures	O2 does not belong to O1. The resulting Architecture would not be fully connected. O2 is the only Network of the Architecture O1.
Description	O2 is removed from the set of Networks of the Architecture O1.

complex_architecture

Input	O: Object
Return	Boolean value
Failures	none
Description	It return true if O is a complex Architecture, nil otherwise.