The DAME Editor: A User Interface for Data Acquisition in an Expert Microprocessor-Based-Systems Designer

by

Dongni Li
B.Eng., University of Science and Technology of China, 1983

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We accept this thesis as conforming
to the required standard

[redacted]

Dr. N.J. Dimopoulos, Supervisor
(Department of Electrical and Computer Engineering)

[redacted]

Dr. K.F. Li, Departmental Member
(Department of Electrical and Computer Engineering)

[redacted]

Dr. Z. Dong, Outside Member
(Department of Mechanical Engineering)

[redacted]

Dr. Daniel M. Hoffman, External Examiner
(Department of Computer Science)

University of Victoria

Supervisor: Dr. Nikitas J. Dimopoulos

ABSTRACT

The automation of computer hardware design has received considerable attention in recent years. Expert systems, that incorporate explicit domain knowledge into problem-solving programs, have been successfully applied in design problems. DAME (**D**esign **A**utomation of **M**icroprocessor-based systems using an **E**xpert system approach) is a system capable of designing microprocessor-based systems from original specifications. In this thesis, the DAME Editor, a user interface for data acquisition for DAME's component database, is presented.

DAME's knowledge domain is represented by a component model which structures the component information into several abstraction levels to facilitate the design process by breaking down the design task into several phases. Based on the component model, each component in DAME is associated to a template which specifies its structure. The DAME editor assists the user in creating and maintaining the component database, through a user graphical interface.

Examiners:

[redacted]

Dr. Nikitas J. Dimopoulos, Supervisor
(Department of Electrical and Computer Engineering)

[redacted]

Dr. Kin F. Li, Departmental Member
(Department of Electrical and Computer Engineering)

[redacted]

Dr. Zuomin Dong, Outside Member
(Department of Mechanical Engineering)

[redacted]

Dr. Daniel M. Hoffman, External Examiner
(Department of Computer Science)

Contents

List of Figures

Acknowledgements

I would like to thank to all the people who helped me to complete this thesis. I especially want to thank Dr. Nikitas Dimopoulos for his encouragement and patience, as well as for his continuous guidance and correction whose imprint can be found throughout these pages. I thank Dr. Kin Li for his kind assistance. Thanks to members of the DAME team, Marco Escalante, Ben Huber and Rodney Pane, for their support, encouragement and positive input. Also I thank my other fellow grad students for those moments that we altogether shared academically and in recreation, and my friends for their caring and thoughtfulness.

Chapter 1

Knowledge Based Expert Systems.

DAME is an expert system that aims to design microprocessor-based systems. Before getting into the details of DAME, we briefly sketch the basic concepts behind knowledge-based or expert systems.

Since the early 70's, when the first expert systems such as DENDRAL and MYCIN were built [1], the interest in expert systems has kept growing, and the number and type of expert system applications are expanding rapidly [2]. An expert system is said to be a knowledge-intensive program that solves problems normally requiring human expertise [3].

The major components of an expert system are: the knowledge base, the inference engine, and the man-machine interface. The man-machine interface in an expert system includes the user interface and the developer interface. The user interface is aimed at the end users who are the consumers of expertise or experts themselves. The major focus of the user interface is on the knowledge domain, that is to represent the domain in a way that is clear for the user to understand. Also the user interface provides feedback and explanations on user's actions and questions. The developer interface is for the system developers who

are responsible of building, updating and maintaining the expert system. The major focus of the developer interface is not only on the different abstract levels of representation of the domain but also the reasoning processes. Sometimes the distinction between developer and end user is blurred, because for some systems the user may be involved in the development and maintenance too.

Figure 1.1 shows the structure of an expert system. We shall discuss these components in subsequent sections.

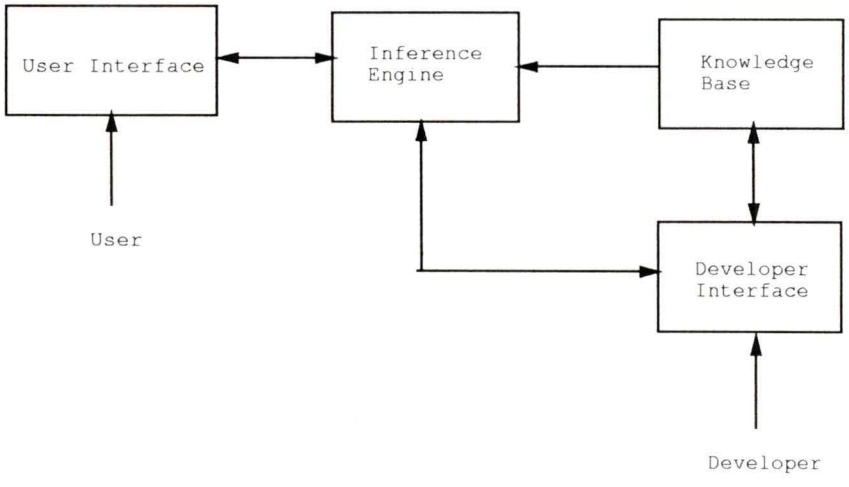

Figure 1.1: The structure of an expert system.

1.1 Knowledge Base

The knowledge base in an expert system needs to store two types of knowledge, the facts and the heuristics. Facts are the explicit and declarative representation of the experts' concepts. Heuristics represent experts' judgmental knowledge. To choose what kind of knowledge to represent and in what form is an important phase during the system design.

There are several commonly used knowledge representations such as production rules, inclusion hierarchies, mathematical logics, frames, scripts, semantic networks, constraints, and relational databases [4]. Among them the production rules and frames are the most popular representations because of their flexibility, representational power, and hierarchical structure. The following subsections explain these two knowledge representations.

The examples in the following subsections are taken from DAME. Here the necessary terminology is introduced informally to aid the examples' understanding.

Microprocessor-based systems are built using off-the-shelf components, such as processors, memory chips, and peripheral devices. These devices are called *components*. A microprocessor-based system is a group of components connected together through the pins of each device. The pins are used to send or receive changing electrical values which are called *signals*.

There are two kinds of signals in a component: the ones that are used to locate and transfer information (address/data); and the ones that are used to facilitate and synchronize the exchange of information (control signals).

While a system operates, certain activities (such as information transfer, arbitration, exception handling, etc.) occur continuously. Not all components are capable of performing all such activities. A *capability* is defined as a primitive activity that a component can carry out. In turn, a component is characterized by the collection of its capabilities. A capability is further specialized through the protocol(s) employed in carrying out the activity associated with it.

1.1.1 Production Rules

Production rules are characterized by two parts: the antecedent and the consequent. The basic form of a production rule is:

```
IF conditions THEN actions
```

For example,

```
IF (component X is an arbiter using a 3-signal protocol)
          AND
   (component Y is a requester using a handshake protocol)
          AND
   (component X and Y need to be interconnected)
          AND
   (there is no interface block between X and Y)
          AND
   (the current phase is the design of the bus-arbitration
              interface)
THEN (create an  instance of the 3-signal to handshake
              interface template)
```

This rule checks if a protocol-conversion interface is required to connect a requester and an arbiter during the bus arbitration interface design phase.

The production rules can be used to represent several types of knowledge, as the following examples show:

Situation/action:	If it is raining, then close the window.
Premise/conclusion:	If it is raining, then the roof becomes wet.
Sufficiency:	If it is raining, then it is precipitating.
Definition:	If it is raining, then water droplets are falling from the sky.

In an expert system, the production rules are used to capture the heuristic knowledge of the domain in a knowledge base.

During the operation of an expert system, a rule is activated (fired) if all its antecedents are satisfied. An inference engine is then used to "drive" the inferencing process. More discussion on inference engines is to be found in section 1.2.

1.1.2 Frames

Production rules are a very good representation for linking conditions with actions. But for representing knowledge about objects, their properties, and the interrelationships of objects in the domain, they are not very convenient and efficient [5]. Frames can be used for structuring information about objects in a knowledge domain in expert systems.

A frame is a data structure containing information about a single entity (i.e., a concept, item, or class). It consists of a set of slots, each of which refers to a specific attribute of the frame entity. A slot contains one or more values of that attribute. We write a frame using the notation:

```
{{ frame-name
    slot1: value1
    slot2: value2
    ...
```

```
}}
```

An example of a frame that describes partially the notion MC68000 (that is the Motorola 68000 microprocessor) is as follows:

```
{{MC68000
    is-a: MICROPROCESSOR
    pins: 40
    package: DIP
    has-signal: MC68000-AS MC68000-DTACK MC68000-D0
                MC68000-D1
    has-capability: DATA-TRANSFER BUS-ARBITRATION
                    INTERRUPT }}
```

The MC68000 notion is represented as a list of attributes/slots. A slot can have a fixed value. For example, the value of PINS is the number 40.

A slot can have a set of attached information such as RANGE which contains the set of possible values for the slot, DEFAULT VALUE, etc. A slot also can have an associated demon which is a program that is triggered when a particular action related to the slot occurs, such as when the slot value is initially inserted, when it is changed, or when it is retrieved.

A slot can also be defined as a relation which can link to other frames. A set of frames can be linked together in a semantic network through relations. The most basic relation between frames is the parent-child relation, in which the parent frame describes a class of items, while the child represents either a subclass (IS-A relation) or a specific member (INSTANCE relation) of the class. For example the MC68000 belongs to the

MICROPROCESSOR class.

There are other possible relations between frames in addition to the subclass and instance relations. Figure 1.2 shows an example of the partial semantic network which describes the Motorola MC68000 microprocessor. The frames are linked together by IS-A, INSTANCE, HAS-CAPABILITY, and USES-PROTOCOL relations.

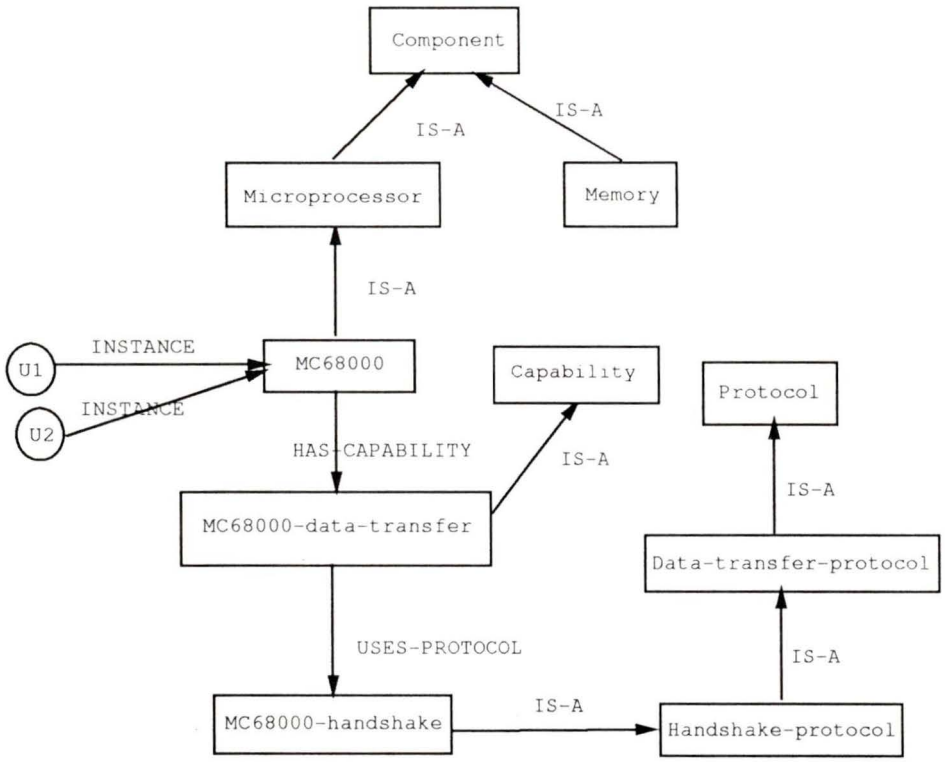

Figure 1.2: Partial semantic network that represents a microprocessor.

A major benefit of utilizing a semantic network is inheritance. The properties and procedures associated with one frame can be inherited by another frame through a relation which links these two frames. In a hierarchical semantic network, the properties and procedures associated with frames high up in the hierarchy are more or less fixed, whereas

frames in the lower levels inherit the slots with values which may be explicitly given or inherited from the parent frames. Thus knowledge can be modeled at different abstraction levels, or modularized in different categories. Not only the storage space of a knowledge base can be reduced by the inheritance of properties, but also the properties of a category can be used where they are needed and the exact location of the properties is not a concern during inference. Also from a development and maintenance point of view, semantic networks can be advantageously used. Because the inherited information does not have to be entered specifically for each individual member of the class, an arbitrary amount of information of a category can be stored at a time and properties can be added or changed easily.

As an example of inheritance, the instance U1 of MC68000 inherits all the information from its parent. However because U1's signals are specific to U1[1], the information in the U1 frame overrides the signal names in the MC68000 frame.

```
{{U1
    instance-of: MC68000
    has-signal: U1-AS U1-DTACK U1-D0 U1-D1
}}
```

The systematic description of the objects and their relationships in a knowledge domain is called the knowledge domain model. An expert system which has a domain knowledge model and a separate rule base is called a model-based reasoning system [6].

A model-based reasoning system which uses frames to implement the domain model is called frame-based reasoning system. In many frame-based reasoning systems, the

[1] in case of several MC68000 chips we would like to name the signals according to each instantiation

information being manipulated is stored in the form of frames, but the problem-solving knowledge that manipulates the frames is stored separately in the form of rules [7].

Knowledge can be classified into shallow knowledge and deep knowledge [8, 5]. The shallow knowledge is represented in terms of heuristic rules, which perform a mapping between data abstractions and solution abstractions. On the other hand, deep knowledge contains information about the causal mechanisms underlying the relationship between the data and the solution. The deep knowledge explores the fundamental attributes of a domain, and represents a more generalised situation.

1.2 Inference Engine

A set of production rules can embody a large collection of expert knowledge. To reproduce part of an expert's reasoning processes, the expert system needs to apply a sequence of rules, which is called chaining. The inference engine is responsible for deciding which rules should be executed, using either a forward-chaining approach or a backward-chaining approach [2].

In forward-chaining, the problem-solving engine operates by selecting one rule that has all its antecedents satisfied to execute at a time. The system initially places a set of input data in the working memory. When the conditions of one rule are matched by the set of data, the rule is said to be activated or triggered. The engine will choose only one rule that is activated and execute it, that is, this rule's action is performed. In other words, this rule is fired. The action of the rule changes the state of the working memory which in turn triggers new rules, that is, the new data will satisfy antecedents of other rules. This select-execute cycle continues until no more rules can be satisfied. Forward chaining is also called data driven or event driven.

In backward-chaining, the problem-solving engine is used to determine a solution. The system initially possesses a set of candidate general solutions. The system considers each solution in turn. For each candidate solution, it finds the rules from the knowledge base that can achieve the solution. If the rule's condition part is true, the solution has been reached. If the condition part is not true, the solution is false. If the condition part is unknown, a sub-solution is set, and then the system tries to prove the sub-solution in a backward fashion until the condition data are known to be either true or false so that the original solution can be proved to be true or false. Backward-chaining is also called goal driven or hypothesis driven.

In both forward and backward chaining, it could be the case that two or more rules are applicable at the same time. The set of applicable rules is called the conflict set. When the conflict set has more than one element, conflict resolution techniques are needed to determine which rule should be executed.

The most common conflict-resolution strategies are: rule ordering, specificity ordering, recency ordering, size ordering, data ordering, context limiting [4]. A system may employ more than one of the strategies listed above as its conflict-resolution technique.

1.3 Man-machine Interface

Expert systems are being used in an expanding number of application areas. The spectrum of expert system applications ranges from very large and complex systems to very small and simple ones. Small, simple expert systems which perform limited but useful tasks may not involve much human intervention, but most other expert systems require high human-computer interaction, at least during the development phase. Expert systems which involve a human user during their operation, require a sophisticated man-machine interface.

As shown in Figure 1.1, there are two types of man-machine interface: one is for supporting system developers, the other is for supporting end users.

1.3.1 Developer Interface

An expert system can be viewed as consisting of a domain-specific knowledge base and a domain-independent inference engine. The developer interface is important when the knowledge base is being developed.

The traditional approach of constructing the knowledge base calls for a knowledge engineer who serves as a bridge between the human expert and the computer, translating the domain knowledge into an appropriate form that can be processed by the computer. This procedure sometimes can be very difficult and time consuming, because it requires the domain expert not only to explain application concepts but also to describe explicitly a decision-making process of which he/she may not normally be conscious [9].

When the model of the knowledge is known in advance, a user interface, through which the user is capable of accessing and manipulating the objects of the model, can help in eliciting and coalescing knowledge. This approach overcomes the above difficulty, and also facilitates the maintenance and updating of the knowledge base when the system is already in use. Instead of interpreting the domain knowledge and constructing the knowledge base, the knowledge engineers must now work with the domain expert to design the domain knowledge model, and write the interface for the user. This interface at the front end provides the high-level knowledge representation language that allows the domain experts to directly manipulate the knowledge, and at the back end transforms the knowledge into the low level representation needed for the computer to be able to process it efficiently. Figure 1.3 shows the difference between the traditional approach and this novel approach.

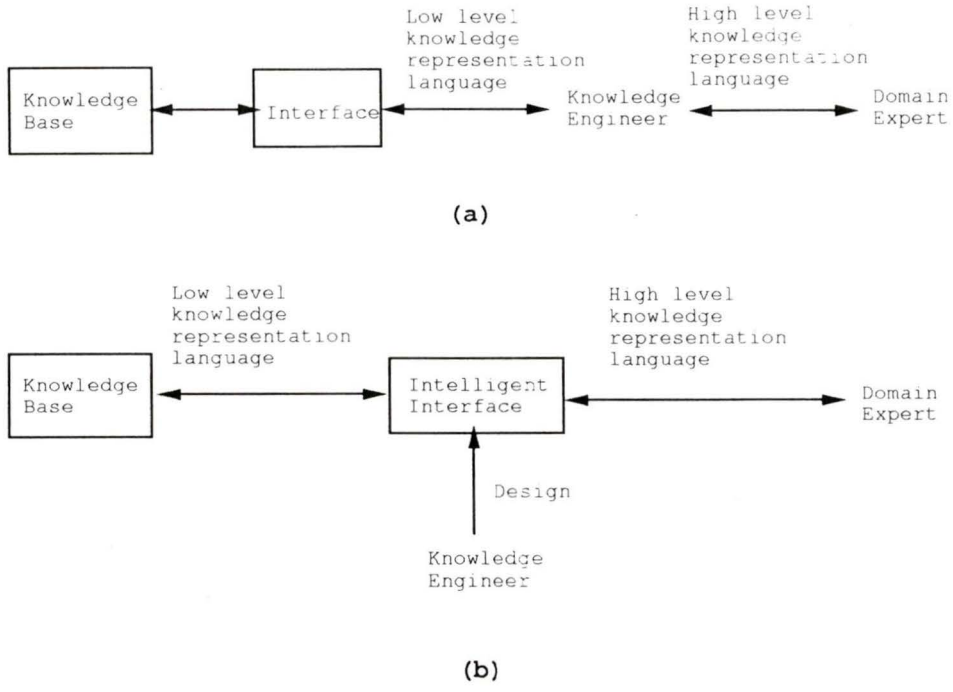

Figure 1.3: (a) Traditional approach of knowledge-base construction; (b) A new approach of knowledge base construction.

1.3.2 End User Interface

Unlike a traditional software system, an expert system is not just a tool that implements a process, but rather a representation of that process [10]. The user of an expert system not only wants to know the result, but also wants to know how the result is reached.

In early expert systems, such as MYCIN and R1, the types of user interface used were conversational, that is, the system asks the user various questions and after going through the inference procedure returns an answer. This type of system is called a consultation system, that is, the user is not involved in the decision-making process. The underlying knowledge of this type of expert systems is considered to be shallow, in which the number of

objects in the domain model is small and the relationships among the objects are simple. As expert systems move from pure consultation systems to expert advisory systems (i.e. both the user and the system share the reasoning and decision-making tasks), the quantity and quality of communication between the system and user need to be increased. Additionally, as the knowledge base becomes larger and the knowledge domain becomes more complex, the use of a deep model of the underlying knowledge is encouraged.

A conversational interface acts as an intermediary to the hidden world of the underlying knowledge and the tasks. It describes the objects and actions of interest to the user and interprets the answers. Contrary to a conversational interface, a model world interface presents the model of the domain and the tasks in a way that matches the user's natural idiom, so that the user can act on the objects of interest directly. A graphical user interface is an example of a model world interface. This style gives the user a better understanding of the task he/she is performing. It also allows the user to directly manipulate the application-related objects and actions which are graphically displayed to him/her, and the system gives immediate feedback on user's actions through the interface. This in turn provides a better understanding of the system's behavior.

1.4 Summary

An expert system can be viewed as being composed of a domain-independent inference engine and a domain-dependent knowledge base. In a model-based expert system, the domain model attempts to give a systematic description of the problem space. The development of an expert system requires the construction of the rule base, and the design of the model of the domain knowledge and the man-machine interfaces for both developers and end users. Model-based reasoning can provide explicit representational support for both interfaces.

While the developers focus on the representation of the domain and the reasoning processes, the end users focus on the domain itself. The developer interface should provide the facility for the developers to look at the representation of the domain, to construct the knowledge base, and to test the system. The end user interface should represent the domain in a way that is easy to understand for the user, and provide immediate response to user's actions and system status.

Although these two interfaces focus on different tasks, there is no clear distinction between developer and end user. The knowledge acquisition tasks which were performed by the knowledge engineer in the past are now often performed by the end user. The developer interface and the end user interface may overlap.

The Objective of this thesis is to present the DAME editor. The DAME editor is a user interface to be used in constructing the component representation used in DAME. Thus, in Chapter 2, we present a summary of several systems capable of automating the design of microprocessor-based systems; in Chapter 3, we discuss issues of knowledge domain modeling in DAME. Chapter 4 presents the editor itself, while chapter 5 presents our efforts in displaying protocols in terms of planar graphs within the DAME editor. Finally we summarize our work and conclude with Chapter 6.

Chapter 2

Designing Microprocessor-based System Using Expert Systems

Hardware synthesis is a difficult task because of the large number of design possibilities which require a large search space. It is a creative work which is based on the expert designer's personal experience and expertise. Since knowledge-based expert systems provide a methodology that can handle real-world, complex problems requiring an expert's interpretation and reasoning, it becomes a suitable approach for high level hardware synthesis. This chapter presents a survey of existing expert systems for system-level hardware synthesis.

2.1 Microprocessor-based System Design

Microprocessor-based system designer uses off-the-shelf component chips to construct computer systems according to the customers' orders or requirements. Designers must

concentrate on system structures, system functions and interconnections among various components and buses. As the number of applications of computers increases, computer systems with various different functions, ranging from multiprocessor workstations to single board systems for instrumentation, are required. Therefore fast and sophisticated system synthesis tools become most important to satisfy the demand for customized designs.

On the other hand, as the library of commercially available components is constantly growing, it is very hard for system designers to keep up with all the knowledge about the available components. Automated synthesis for system configuration and design offers a solution to this dilemma.

R1, MAPLE, MICON, and KMDS are existing systems that produce designs from system specifications. All these systems utilize an expert system paradigm.

2.2 R1

R1 is the first expert system that attempted to solve system configuration problems [11, 12, 13]. The R1 system configures the computer systems Digital Equipment Corporation manufactures.

The task that R1 performs takes a list of components a customer has ordered as its input, and produces a set of diagrams that display the interrelationships among these components as its output.

The knowledge that R1 has is about the pre-designed semi-products (parts) of computer systems, such as a bundle, a Unibus module, or a power supply, and how to put them together. R1 is considered a shallow model system. The objects it has in its knowledge domain are components and cabinet templates. Components are used to describe semi-products of the

VAX systems. Each component description consists of several attribute/value pairs that indicate the properties of that component that are relevant for the configuration task. A cabinet template describes what space is available in a particular cabinet type. It describes the physical space of the cabinet and locations of the components that should be filled in. These templates serve as a guide for R1 to know at any point in the configuration process, what container space is still available, and where the specific location is.

R1 was originally implemented in OPS4 and uses a context limiting conflict-resolution strategy. Its configuration rules are constructed in six subtask contexts:

1. Checks the customer's order.
2. Configures cpu cabinet.
3. Configures the UNIBUS cabinets.
4. Labels the UNIBUS expansion cabinets.
5. Designs the system and lays out its diagram.
6. Designs the cabling.

Figure 2.1 shows the original R1 system structure. It consists of a database which contains the components and cabinet templates, a rule base, a conventional user interface, and a backward chaining inference engine.

R1 was a success. The system has been extended to perform all the configuration tasks for all kinds of computer systems that Digital Equipment Corporation manufactures. The biggest problem it confronted was the continuous expansion of the knowledge base. In five years its rule base has been expanded from 772 rules to 4,000 rules, and the data base has grown from containing 240 components to 10,000 components [14]. The maintenance and

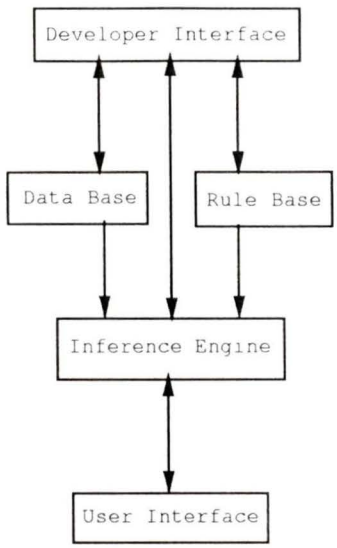

Figure 2.1: R1 system structure

continuous development of the knowledge base became substantially difficult. The main difficulty is that when an expert or knowledge engineer adds a new piece of configuration knowledge to the rule base, it is very hard to identify the role of this piece of new knowledge; in other words, it is hard to know how it will effect the rest of the rules. In order to resolve this problem, the R1 designers reconstructed R1's rule-base. They divided each of the subtask contexts into four contexts. They are: evaluate goal context, propose-operator context, evaluate-operator context and apply-operator context. In this way each piece of knowledge has an explicit role.

After reconstructing R1's rule-base, the R1 designers built a knowledge acquisition subsystem to assist experts in adding new knowledge to the knowledge base. This later version of R1 is called Proto-R2, and it is implemented in OPS5. The acquisition subsystem is called Sear. Figure 2.2 shows the system structure.

Sear's major components are an interviewer and a rule generator. The interviewer

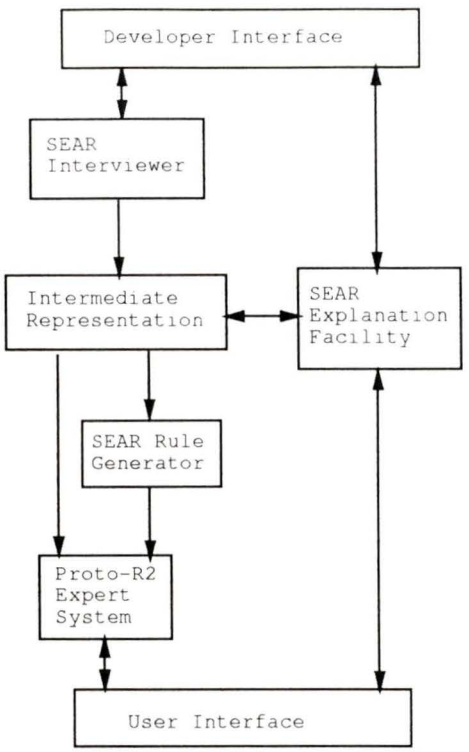

Figure 2.2: R2 system structure

elicits domain knowledge either from a knowledge engineer or a domain expert, translates it into an intermediate representation; then the rule generator converts the intermediate representation into OPS5 rules.

2.3 MAPLE

MAPLE (Microprocessor ApPlications Expert) is a consultational expert system which designs hardware for microprocessor applications [15].

MAPLE uses the case-based reasoning approach. Unlike rule-based reasoning which

solves problems by taking an input specification and then "chaining" together the appropriate set of rules from the rule base to arrive at a solution, case-based reasoning solves problems by searching its case memory for an existing case that matches the input specification [16]. If the input specification matches an existing case, the system gives a solution. If the there is no exact match, the system chooses a similar case and modifies a small portion of the case and its solution to meet the input specification. The adjusted case becomes a new case saved in the case base.

MAPLE takes as input a set of hardware specifications and constrains, and gives as output a design report which includes the original specifications and a set of complete documentation for the design.

MAPLE has three databases:

- a BOARD database which stores pre-designed boards,
- a REPORT database which stores the system's previous design experiences.
- and a COMPONENT database which stores the information about the basic devices such as CPU, memory, I/O, and discrete logic.

MAPLE's design procedure includes three stages: interview, design and report. During the interview stage, the user is asked through a conventional user interface to give the hardware requirements and constraints. During the design stage, the system first looks for the combination of previously designed boards from the BOARD database which satisfies the requirements and constraints. If there is a match, the design is done, otherwise a description of the requirement is retained and suggested to the system manager for possible design as new board. During the report stage, only when the design is a combination of pre-designed boards, a report is selected from the REPORT database.

Ideally MAPLE would be able to design new boards automatically. The current MAPLE is not involved in any board level design. If MAPLE can not find any previous design which satisfies the current specifications, it suggests to the experts to give a new design and stores the design in the BOARD database and the documentation in the REPORT database.

2.4 MICON

MICON (MIcroprocessor CONfigurer) is a system used to configure microprocessor-based single board computer systems [17, 18, 19, 20].

The construction of single board computers is based on the integration of fixed subsystems or functional units such as memory arrays, processors, and peripheral devices and controllers. The task of designing single-board computers can be viewed as selecting the required functional units and interconnecting them. MICON breaks down the design process into a set of tasks and associated subtasks that correspond to the construction of these functional units. The set of design tasks are: processor design, memory design and peripheral design. Each design task consists of three steps: specification, selection and instantiation. During the specification step, the user is requested to enter requirements for a particular subsystem. After the requirements have been ascertained, the system does the selection. The required parts are sought in the database; if no part exactly matches the requirements, the closest alternative is substituted; if there is no alternative, the user is notified and must change his/her requirements. During the instantiation step, the selected parts are instantiated into the design. This is done by creating "logical" copies of the devices from the database and interconnecting them.

The knowledge of how to map the newly instantiated parts into the evolving design is given in a set of rules called templates. Each template is associated with an implementation

technique. The left hand side of a template rule is a set of preconditions which identifies when the template should be applied. The right hand side of the template rule contains a version of the wire list.

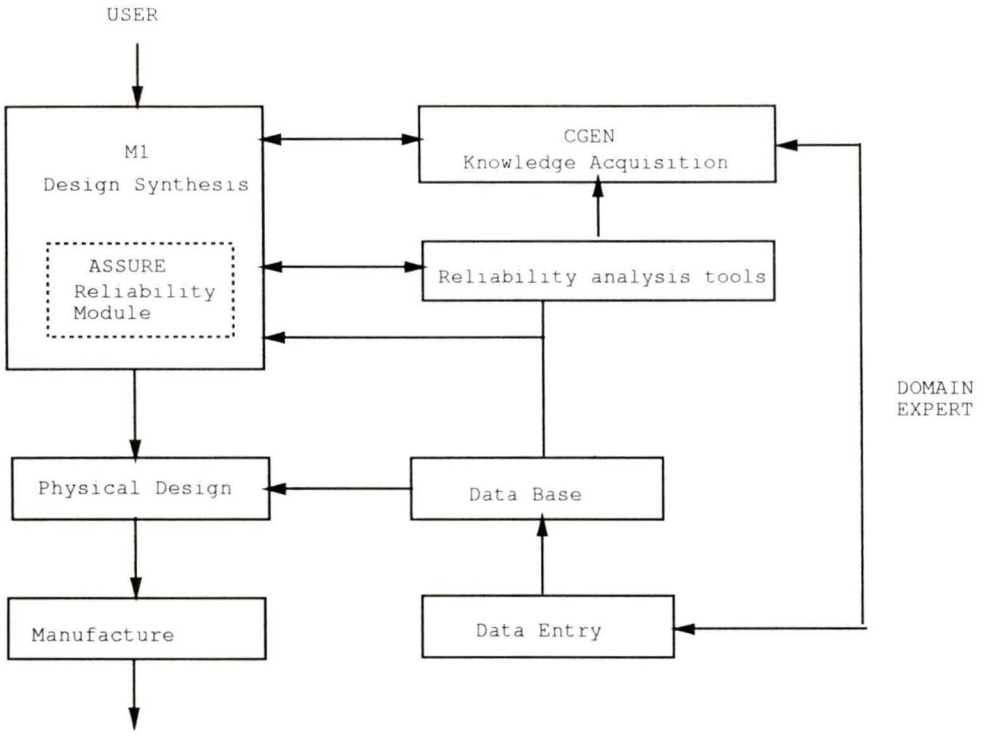

Figure 2.3: MICON system configuration

MICON is an integrated computer design system. Figure 2.3 shows the complete MICON system. MICON's database consists of the functional hierarchy of a computer system and the description of each part. MICON is implemented in OPS/83. Its rule-base consists of templates and other rules that support other design aspects.

CGEN is a knowledge acquisition subsystem associated with MICON. Whenever, during the instantiation design stage, the system cannot find an appropriate template to map

the current functional unit into the evolving design, it reports this situation to CGEN. The domain expert gives a mapping template for this case, and saves it in the rule-base.

During the design task, there are three subtasks:

1. Processor design. The system asks questions such as processor, application, cost and power, etc. Afterwards the system will choose a processor which most closely matches the requirements.

2. Memory design. The types of memory (SRAM, DRAM, ROM) and the amount of addressable memory are specified by the user. If the amount of specified memory exceeds the addressing range of the processor, the user is notified and the requirements must be changed by the user. After the type of memory chips has been selected, the array size is determined and the chips are logically interconnected. The inclusion of any support chips, usually in the form of memory controllers, occurs automatically without the intervention of the designer.

3. Peripheral design. The specification of peripheral devices is slightly more complicated than the previously mentioned subsystems. Peripherals are specified after the memory, so that certain parts can be shared between subsystems. Typically these parts are chip select logic and bus drivers. The peripheral devices could be timers, PIO (parallel input-output device), SIO (serial input-output device) or DMA (direct memory access) controllers. After the appropriate device has been found in the data base, MICON assigns an address and an interrupt priority to the device according to the user's specification or the system default. Finally the chips are instantiated into the design by interconnecting the interrupt lines followed by the data, address, and control lines.

Each design subtask is activated at some time during the design process, and it consists

of three steps.

1. Specification: the user is requested to enter requirements for a particular subsystem;
2. Selection: after the requirements have been ascertained, the required parts are sought in the data base; if no part exactly matches the requirements, the closest alternative is substituted; if there is no alternative, the user is notified and must change his/her requirements;
3. Instantiation: once parts have been selected, they are instantiated into the design, this is done by creating "logical" copies of the devices from the data base and interconnecting them.

The other task of MICON is the analysis of the design. The purpose of this task is to allow the user to analyze what MICON has produced. This task also provides the user with the capability to verify part of the design or modify a portion of the design.

In MICON, most of its design knowledge is stored in templates. The major templates describe the relationship between subsystems, and are used during the inter-subsystem design. There are three abstract levels of templates:

- level 0: contains the general description of the subsystems comprising a single board computer system.
- level 1: contains the general representation of the single board computer system being designed with respect to a processor family.
- level 2: contains implementation details for individual subsystem components. The templates are represented as amalgamation of parts and ports. All the parts in a

template are represented as part-descriptors which are actually chip elements. All ports represent the connections between these parts.

2.5 KMDS

KMDS is an expert system for integrated hardware/software design of microprocessor-based digital systems [21]. Figure 2.4 shows the system configuration. The major modules are a menu-driven user interface for obtaining user requirements and constraints, a user demand and solution analysis module, a logic circuit synthesizer, a control program generator, and a knowledge-base debugger for updating and maintaining the consistency of the knowledge-base. Other modules in the system, such as the scheduler, an LSI (large scale integrated circuit) device library, an AUX IC (auxiliary integrated circuit) device library and other external utilities, are used to control and support the design procedure. The LSI device library stores the physical, behavioral, and structural information of components and also their programming information. The AUX IC library stores the information about auxiliary devices such as gates, decoders, multiplexers, etc.

KMDS uses a frame structure to represent its problem model. The frames are organized into trees. A root frame represents a system. The system is divided into several modules, such as processor, memory, I/O. Each one is represented by a frame. These modules are in turn divided into submodules and smaller submodules until the primitive functional units are reached. There are two types of frames in KMDS: packed frames and primitive frames. A packed frame has three kinds of slots: (1) specification slots, which contain associated specifications of the hardware modules represented by the frame; (2) construction slots, which describe the internal structure of the hardware module; and (3) a signal pool slot, which contains all signal information of the frame. The values in a construction slot are

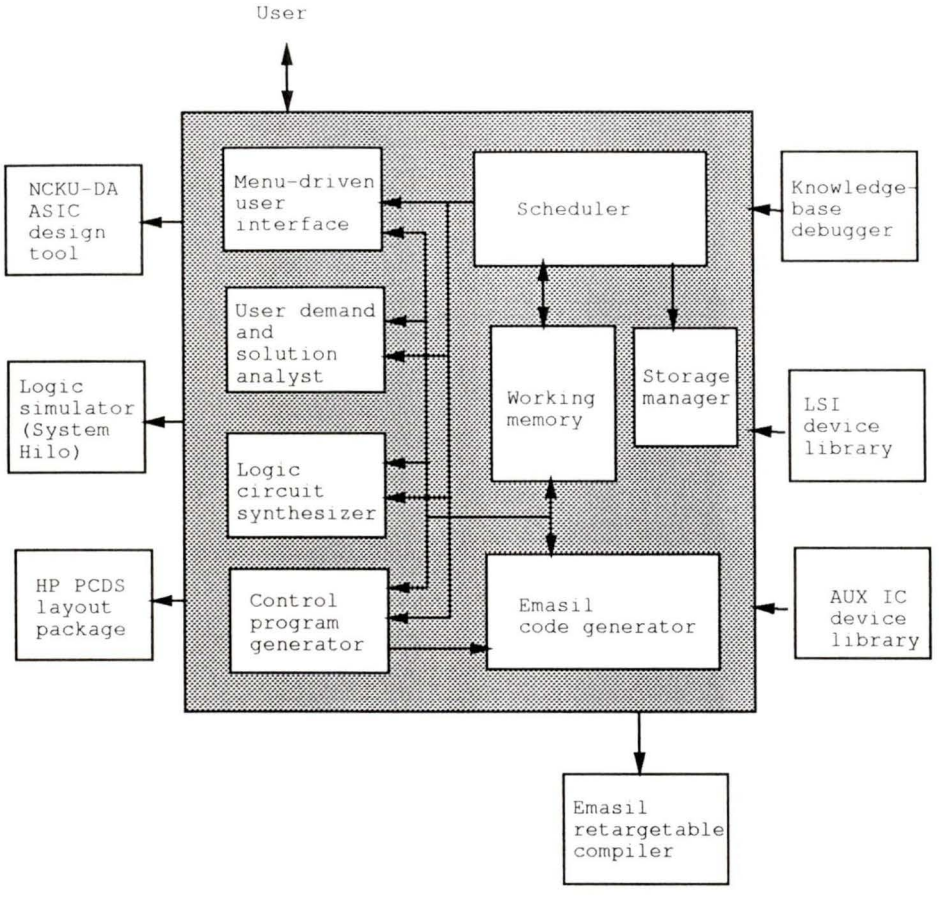

Figure 2.4: KMDS system configuration

names of other packed frames or primitive frames which represent submodules of the frame. Each frame can only have one signal pool slot.

A primitive frame can only have specification slots and a signal pool slot, because it represents a primitive function unit.

The following is an example of a packed frame for the 63484 CRT interface module:

Frame name: 63484-CRT-interface-module

1. CRTC series no: (63484ps4/63484ps6/...)
2. CRTC frame: (CRTC63484)
3. Buffer: (FM8/FM16)
4. Data bus width: (8/16)
5. Bits/pixel: (1/2/4)
6. Maximum: (columns, rows)
7. Signal-pool: (S1, S2, ...)

In this example, slots 1, 2 and 3 are construction slots, slot 4, 5 and 6 are specification slots, and slot 7 is the signal pool slot.

KMDS's design procedure proceeds in three phases. After the system obtains reasonable and complete design requirements and constraints from the user, the first design phase, user demand analysis, is invoked. It translates the user requirements into design specifications and constraints, then constructs the architecture frame. If no architecture satisfies the user's requirements, it will ask for new requirements.

The second phase is the hardware design phase. This is done by the circuit synthesizer module. First, according to the architecture frame, it generates the needed construction slots, their values and the signal connections recursively until it achieves a set of primitive frames that can construct the selected architecture frame. Then it uses the information in the LSI device library and the AUX IC library to realize the physical design and obtain the final circuit. Finally it generates a net-list file written in EDIF (Electronic Design Interchange Format).

The third phase is software design. The control program generator can automatically generate the control routines of the synthesized digital system written in Emasil (which is a Processor-Memory-Switch level language). The Emasil retargetable compiler can translate the control routines into the machine codes of the selected microprocessor.

2.6 Summary

Several systems which tackle the design of computer systems have been described in this chapter.

R1 is a land mark in that it introduced a new methodology in the design. A top-down approach is followed by R1 through the traversing of stages that incrementally complete the configuration of a computer system. The primitive building blocks in R1 are modules, racks, etc.. However our main interest here is the design of Microprocessor-based systems, a finer-grained problem.

MAPLE introduces the concept of adapting a previous design to come up with novel/new solutions. The primitive building blocks in MAPLE are single-board computer systems. Case-based reasoning saves time in producing designs that belong to the same class. But additional artifacts are needed to accommodate new classes of designs.

MICON's design approach starts from high-level requirements to design a single-board computer system. CPU selection, memory selection and I/O selection are addressed sequentially. Primitive building blocks are templates: a component together with the necessary interfacing logic. A design consists of the instantiation of templates so that the interconnection is direct.

KMDS is similar in its approach to MICON but adds to the design process the software

control routines. However the application of KMDS has been limited to the design of single board microcomputers and CRT controllers.

All the systems covered here are expert system designers. The objective of a design problem is to construct a system or object satisfying a given specification. The components are the primitive objects from which the construction is to be done. Construction is done by applying selection and connection of components operations [20]. In this chapter we have seen that different systems attack the representation problem at different level of detail. In DAME we aim to show that representing the components at a finer level allows us to design with more general design knowledge.

Chapter 3

DAME Knowledge Domain Modeling.

3.1 Introduction

DAME (Design Automation of Microprocessor-based systems, using an Expert system approach) is an expert system which will be able to synthesize a customized microprocessor-based system from user's system specifications [22, 23].

Figure 3.1 shows the DAME system configuration modules. It consists of a window-based mouse driven user interface, an intelligent knowledge editor, a component library, a working memory and five design modules which represent the five design phases.

The system specification module is used to analyse the requirements and constraints given by the user's input specification, and to transform them into a complete and consistent design specification. After the system specification has been established, the system

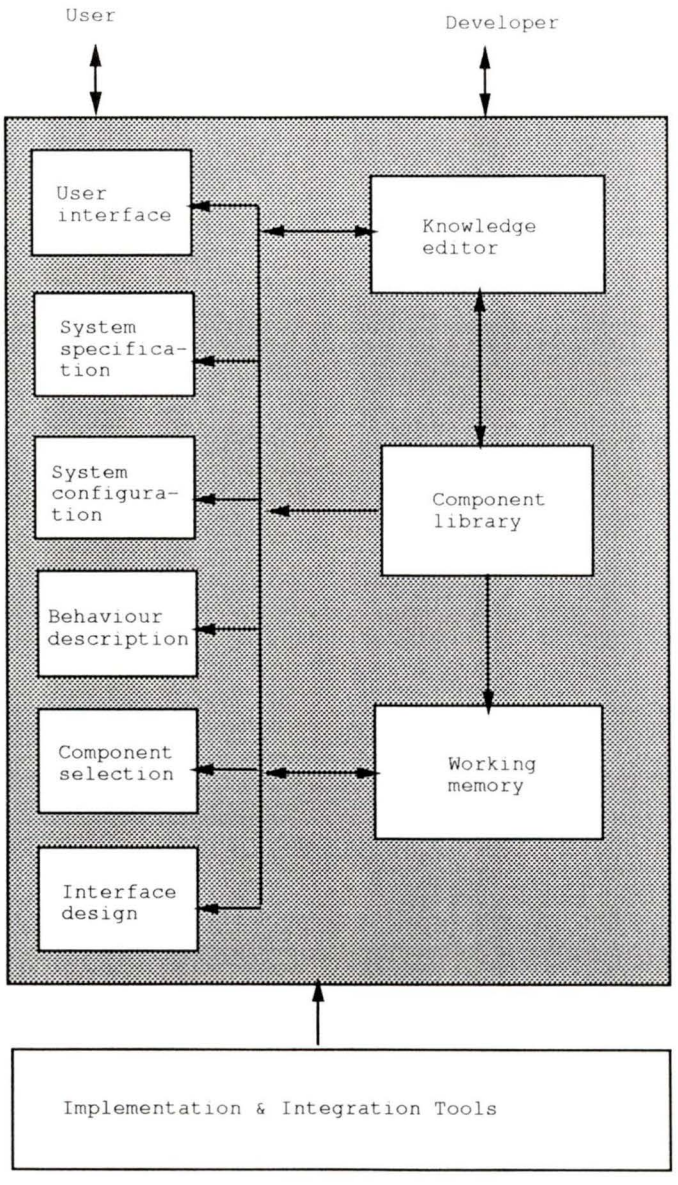

Figure 3.1: DAME system structure

configuration module constructs an architecture frame which consists of several interrelated subsystems according to the system specification. The behavior description module defines each subsystem's capabilities and constraints. The component selection module then maps each subsystem into the available components. Finally the interface design module connects the components together and generates the necessary glue logic.

DAME is implemented in Knowledge Craft, which is an integral expert system shell consisting of inference engine, working memory, editing and debugging tools, and window/graphic facilities. DAME uses Knowledge Craft's CRL-OPS work-center as its inference engine and working memory.

As mentioned in the previous chapter, DAME aims to model the components at a finer level so that more general design knowledge can be applied. In MICON, the connection of two components is pre-designed and stored in templates. During the design process, if the system cannot find the template required to connect the two components, it reports to the design expert and waits until the correspondent template is added into the knowledge base. Only then can the design be continued. As new devices are announced, the knowledge base has to be expanded with new design knowledge advances.

DAME aims to carry out the interface design based on the interfacing protocols used by the components to be interconnected. In this sense DAME is capturing the experts' knowledge of how to connect two components. Although there are numerous microcomputer devices, only a few communication protocols are used by the components. Thus few general rules are necessary to accomplish the design.

This chapter will introduce DAME's component model.

3.2 Component Model

When we study a component, there are three kinds of information that are important at different stages of the design: general component information, signal definition, and interface specification and timing constraints. General component information describes static features about the component, such as device name, manufacturer, cost, temperature range and reliability, speed of operation, package type available, power requirements and consumption. The signal definition describes the characteristics of signals, such as signal name, direction(input, output, bidirectional), pin number, active status. In the interface specification and timing constraints, it is described how the component communicates with other components.

This information is needed throughout the design process, but at each design phase only part of the information is important for the design to advance. As the design process proceeds, more detailed information is required. In DAME a component is modeled hierarchically. Different levels of information abstraction are encapsulated at different levels in the hierarchy of design objects in the semantic network representing the component.

The component model consists of four abstract levels and five types of objects. These objects are component, signal, capability, protocol and action. These objects are interrelated through relations. The four relations are has-signal, has-capability, uses-protocol, and has-action. Figure 3.2 shows a conceptual component model.

An object is represented by a frame or schema. A schema can have many attributes called slots. A relation is a specially defined slot whose values are other schemata. The definition of a relational slot is another schema which has the same name as the slot. In the definition schema, the properties of the relation, such as range, domain, inheritance, etc., are specified. For example, the definition of the has-signal relation specifies that the range,

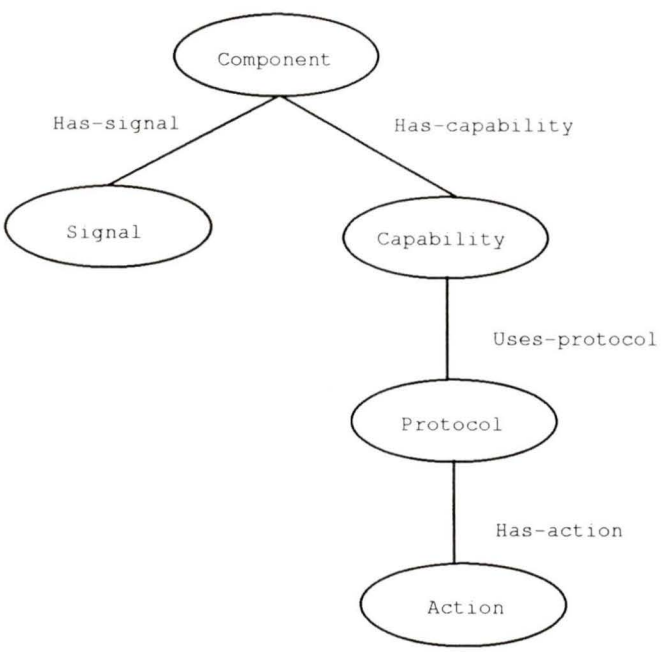

Figure 3.2: DAME component model

which is the object the relation links from, has to be a component object, and the domain, which is the object the relation links to, has to be a signal object. Thus if a value which is not a signal object is given to the has-signal slot, an error is displayed immediately.

The following subsections will discuss the five types of objects in the component model.

3.2.1 The Component

The component object is at the highest level. It contains the general specification information of a component, and is linked to the signal and capability objects through the has-signal and has-capability relations, respectively.

3.2.2 The Signal

A signal object stores the signal's definition information, i.e., a signal's name, direction, pin number, active status, etc. One signal object represents only one signal of a component. Therefore if a component has N signals, there will be N signal objects attached to the component object.

3.2.3 The Capability

A component can communicate with one or more types of components according to its functionality. A capability defines the kind of interfacing ability a component has. A component can have one or several capabilities. For instance, a memory component has only data transfer capabilities; a peripheral device may have data transfer and interrupt capabilities; while a microprocessor should have not only data transfer and interrupt capabilities in order to be able to interface memory and I/O components, but also a bus arbitration

capability which allows it to compete for the use of the bus in a multi-master system.

3.2.4 The Protocol

Capabilities tell what a component can do. Protocols define how a capability is exercised. DAME is based on the assumption that there are but a few protocols which can be used by the various capabilities. There are three basic types of protocols: synchronous, handshake, and semi-synchronous [24]. A protocol is a sequence of signal actions that guarantees the proper function of the capability. It is assumed that, at any moment, only two components actively transfer information between each other over an information exchange path which may be a dedicated point-to-point connection between the two components, or it may be shared with other components as well.

For instance, the CPU may read/write data from/to memory over a common bus which may be shared by several peripherals.

A synchronous protocol sequence is defined as follows: data is presented by the source and held stable for a limited period of time. Explicit (through strobes) or implicit data annunciation is provided. In the explicit annunciation, when the destination receives the strobe signal it reads the data in the strobe assertion period.

In a handshake protocol cycle, also called an asynchronous protocol, the source also sends data and strobe signals, but instead of waiting for a fixed time, it waits until the destination issues an accept signal.

A semi-synchronous protocol cycle is similar to the synchronous protocol: after sending data, signalling the strobe, and waiting for a fixed period of time, it checks a special no-accept signal. If the destination does not issue a no-accept signal, the sequence proceeds; otherwise the source waits until the no-accept signal disappears.

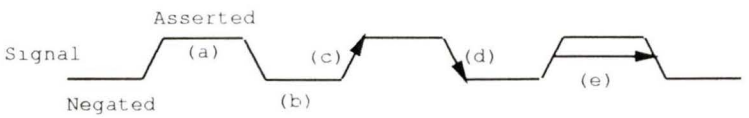

Figure 3.3: Signal encoding actions: (a) asserted, (b) negated, (c) negated to asserted, (d) asserted to negated, (e) duration.

3.2.5 The Action

Microprocessor components transfer information in the form of signals through wires that connect them. A protocol enforces the correct transfer of information by defining the order and timing of elementary operations. The elementary operations in the protocol are called actions. Actions are encoded by signal states and transitions.

For example, a signal can transfer information by its static levels (asserted, negated), dynamic changes (negated to asserted, asserted to negated), or its duration between two changes (see Figure 3.3). An action can be defined by the signal transitions or states, or combinations thereof.

The behaviour of a protocol can be captured from its timing diagram using an action graph [25]. The definition of an action graph is given as: let **A** be the set of actions in the protocol, **P**(for precedes) is the relation for any two actions $a, b \in A$, such that $(a, b) \in \mathbf{P}$ *iff* action b is preceded by action a in the protocol cycle. A directed graph (**A**,**P**,t_p) can be used to represent the protocol, where **A** is the set of vertices of the graph, **P** is the set of edges, and t_p is a function on **P** that associates a label (t_{min},t_{max}), with the minimum and maximum timing between actions, to the edges. For self-timed circuits, t_p is $(0, \infty)$.

Figure 3.4 shows timing diagram for a full handshake protocol. In Figure 3.4 signal R is an output signal and signal A is an input signal from the source. The assertion of the

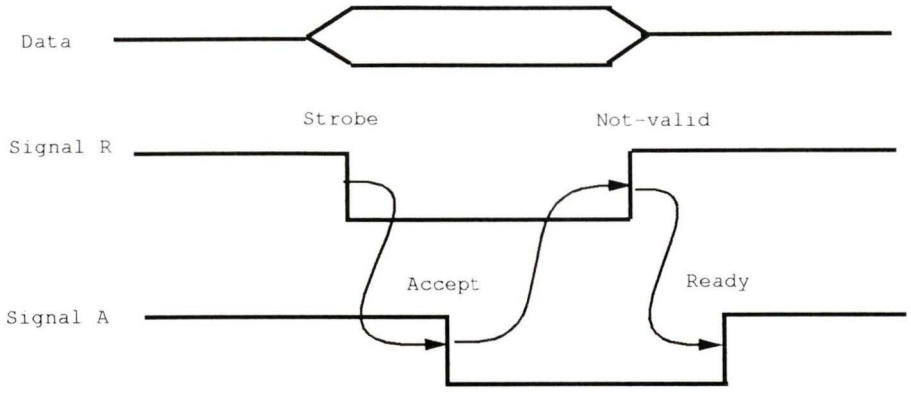

Figure 3.4: Handshake protocol: signal representation

signal R indicates the strobe action and the negation of signal R corresponds to a not-valid action. Similarly, the assertion of the signal A indicates the accept action and the negation of signal A corresponds to the ready action. After the data is presented at the data bus, the source signals the destination by the strobe action that the data is sent, the destination responds with an accept action that the data is received, then the source tells the destination that the data is not valid any more by the not-valid action, and the destination responds with a ready action meaning it is ready for further communication. This full handshake protocol with four actions has the advantage that the signals return to their initial state by adding two extra actions as mentioned in the previous subsection. The corresponding action graph is shown in Figure 3.5.

3.3 Component Templates

In the previous sections of this chapter we have discussed DAME's component model in terms of capabilities, protocols and actions. A component template is a network of schemata which describes the invariant features of a component class based on the component model

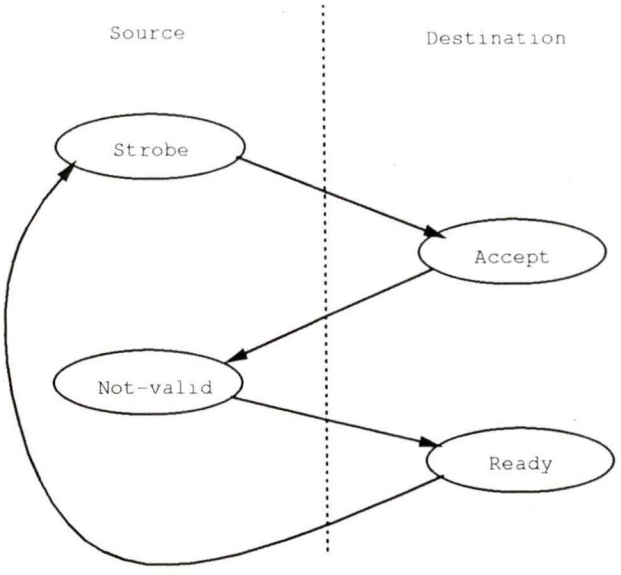

Figure 3.5: Handshake protocol: graph representation

(i.e. the capabilities the class of components has, protocols it uses, etc.).

One can think of a template as an empty structure resembling actual components, ready to be filled in with the specific information that changes from one component to another, such as the name of the signals, specific protocols, etc. On the other hand, certain assumptions predicate the existence of certain objects of the component. For example, it is assumed that any component is always capable of exchanging information, thus a data transfer capability is always present. The data transfer capability can be further divided in to read, write, or read and write capabilities depends on the type of the component. As an example, a ROM has only a read capability. The basic component classes are memory, I/O device, microprocessor, and bus. Each class may be composed of several subclasses.

Figure 3.6 shows the microprocessor template. It consists of signals as well as four capabilities (i.e. bus arbitration capability, data transfer capability, interrupt capability and

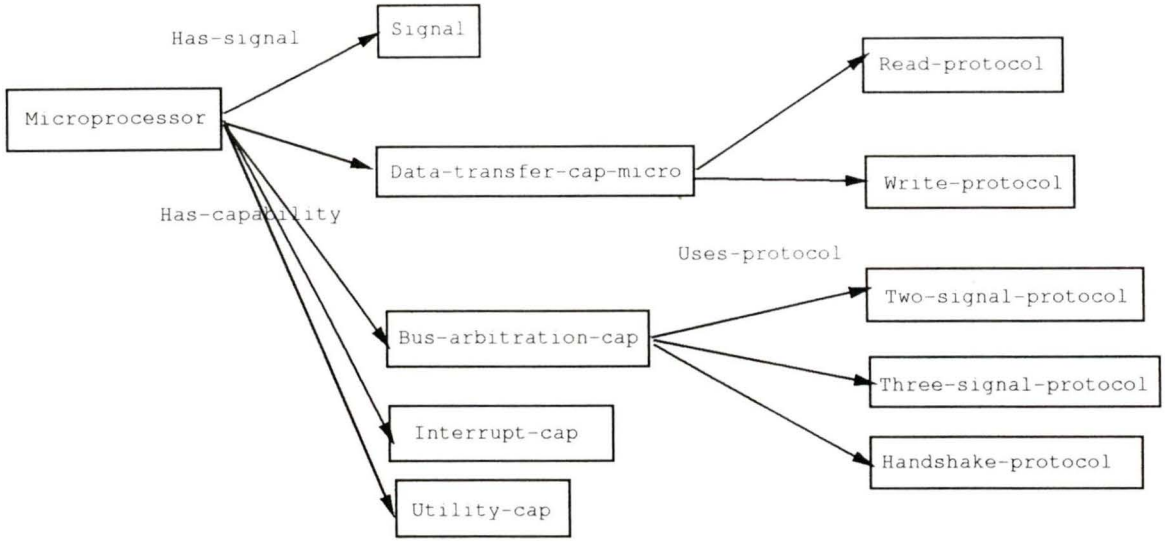

Figure 3.6: The microprocessor template.

utility capability). The microprocessor's data transfer capability is also represented as a template which includes details of the read and write protocols. Similarly the bus arbitration capability incorporates the details of the bus arbitration protocols. These protocols are selected from standard basic protocols that components use to communicate with each other.

Figure 3.7 shows the template for the memory component. It has a similar structure as the microprocessor template, but because a memory cannot function as a master, it only communicates through the data transfer capability.

A particular component uses the generic template defined for its class, instantiated with the specific information about the protocols and signals which are used by that component. Figure 3.8 shows the semantic network representing the MC68000 microprocessor component. It has the same structure as the microprocessor template, but contains the signals and protocols that the MC68000 uses. Figure 3.9 shows the network for the MK6116 memory

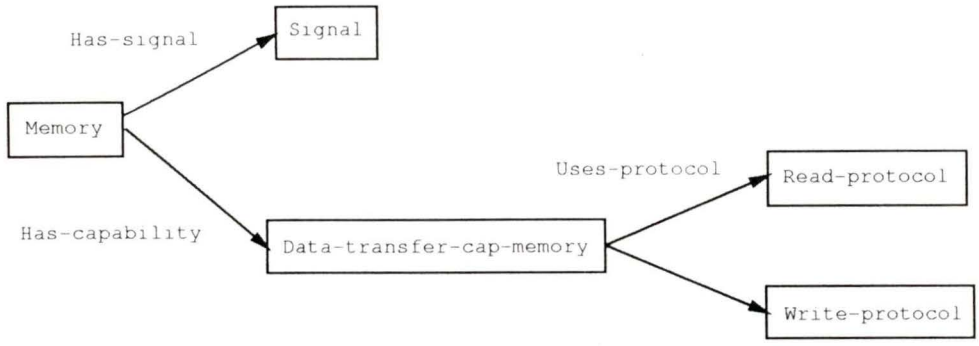

Figure 3.7: The memory template.

component.

3.4 Summary

In this chapter, the component model used by DAME to represent components in the database was introduced. This component model interacts with the design knowledge (i.e. the rules) in DAME following a top-down approach which concentrates on the important information at each level of the design. The top levels of the design process look at the top layers in the component structure such as the basic classes and sub-classes of components. For example, a system for multi-media may require a specialized DSP with associated RAM and ROM memory for one of its subsystems. In the component selection phase, only the information about the actual components such as the DSP Motorola 96000 and the RAM chip MK6116 is considered. After the physical components have been chosen, DAME designs the interface taking into account each capability. In a simple example of an interface between a CPU and a memory device, only the data transfer capability has to be considered to design the interface. The interface block will be designed according to the data transfer protocols each component uses. At the end, the real signals are annotated

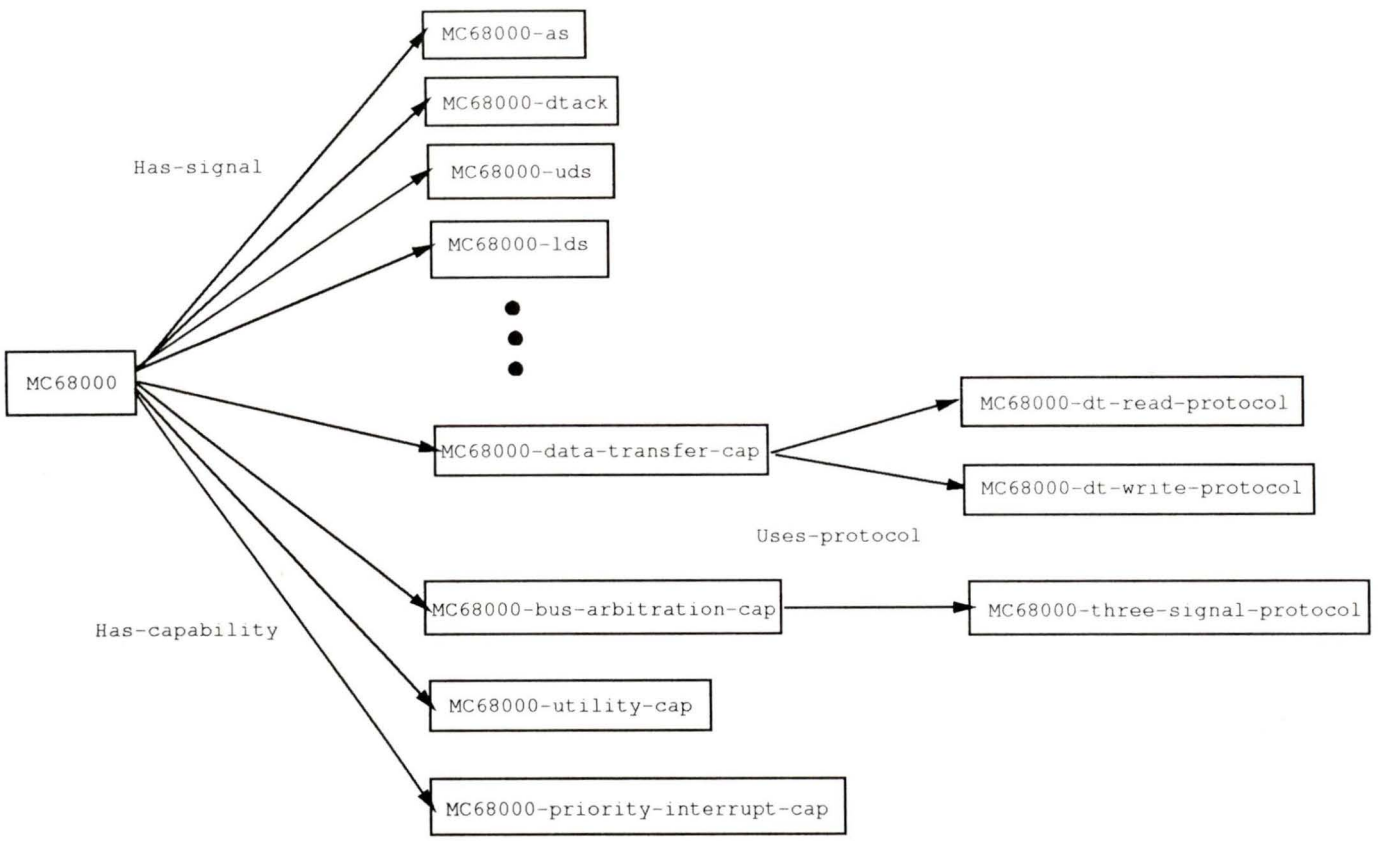

Figure 3.8: The MC68000 component

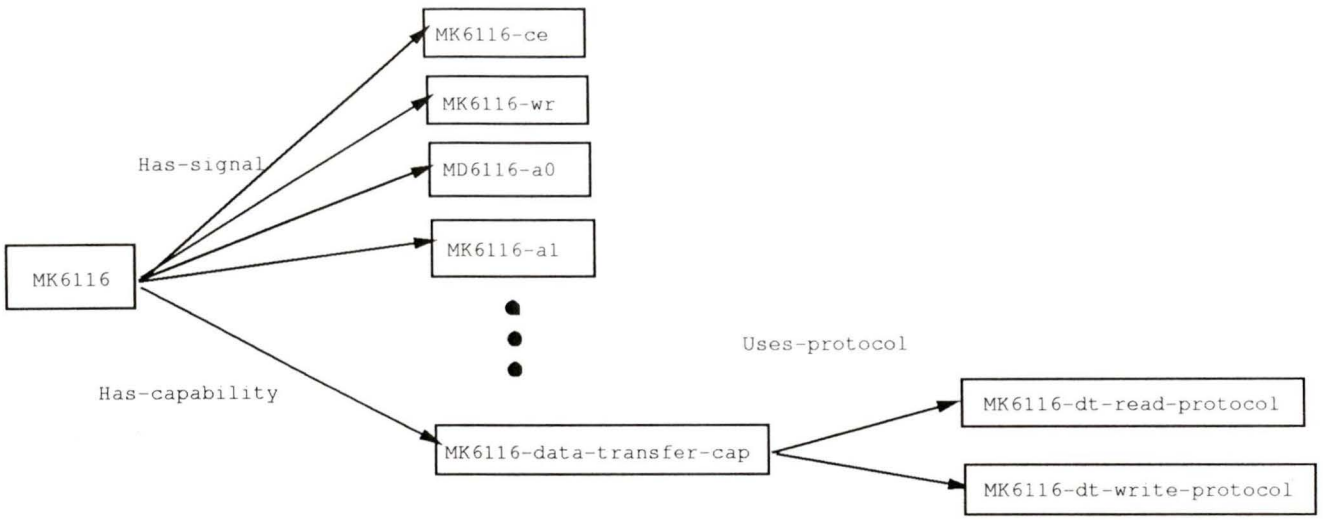

Figure 3.9: The MK6116 component.

back to the interface block description.

The component templates and instances were also introduced in this chapter. A network of templates structures the component information into hierarchical levels. Each component instance is not only created using the same data structure as its template, but is also linked to its template so that it can inherit the features shared by all objects belonging to the same class. In this manner, the component library can be easily built by copying the template structure. DAME's editor has among its tasks the addition of new components into the component database. In the next chapter DAME's editor is presented.

Chapter 4

DAME Editor

4.1 Introduction

DAME uses a semantic network [4] to describe components and their properties. Each component in the component library contains the same data structure as its template. The DAME editor utilizes component templates to assist the user in creating and modifying the component library. Additionally, it presents components in a way that is easy for the user to understand and modify, and allows modifications of the component library structure that are incorporated into the frame-based semantic network.

The DAME editor uses a mouse driven graphical user interface. It is implemented on top of the Knowledge Craft's Graphics, Window system and Command system modules. It resides in the DAME shell, a developer interface which has access to several modules from Knowledge Craft. These are: a schema editor (P-Editor), a schema network editor (Palm Editor), the CRL-OPS work centre, the KC Listener, and the Toolbox. Figure 4.1 shows the Knowledge Craft module hierarchy.

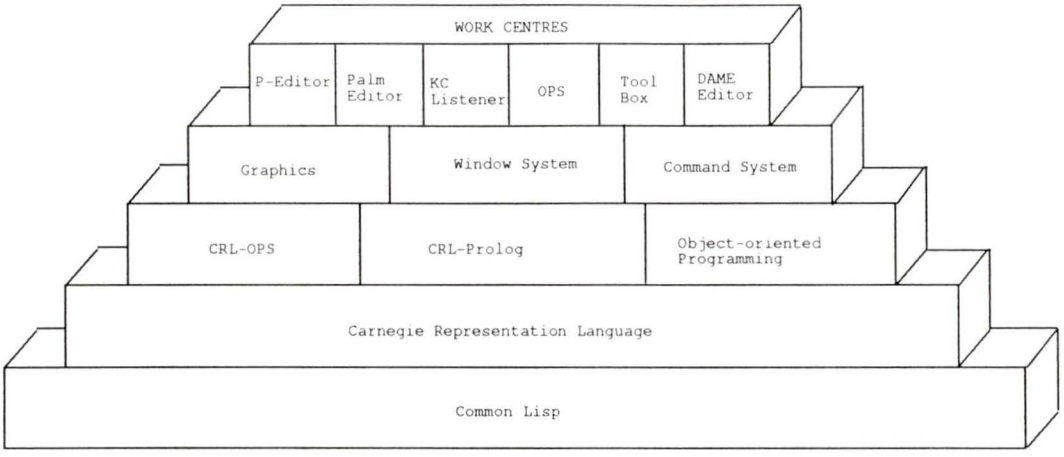

Figure 4.1: Knowledge Craft modules

The DAME editor implements the following functions:

1. Create a new component,
2. Edit component,
3. Access the component library (save component, load component), and
4. Display a protocol graphically.

This chapter will first introduce the copy template function which is the key function used to add new components to the library. The features of the editor are then explained in detail.

4.2 The Copy Template Function

A component instance has a structure similar to its template. For example compare Figure 3.8 and Figure 3.6. The portion of the MC68000 data structure that is preserved from its

template is the capability level. A component always has the same number of capabilities as its template. Signals, on the other hand, are specific to the component. Each capability in a template is linked to all the possible protocols it may use. The protocols used in the component's capabilities are a subset of those in its template.

When a component instance is created, a portion of the template structure is copied while other parts are filled in with information provided by the user. Figure 4.2 shows the partial structure copied from the microprocessor template that is the basis of the MC68000 component. The protocols used by each capability are chosen by the user from a list of relevant alternatives, while the signals are entered by the user.

The copying of template structures is performed by the copy-template function. It is a recursive function which copies a schema from the template to another schema in the component instance. A schema in a template consists of a set of non-relational slots, each of which may or may not have values, and a set of relational slots, each of which has a value: the name of another schema. The copy-template function copies all the non-relational slots along with their values to the new schema, and copies the relational slots and either leaves a relational slot's empty (without any value) or makes copies of its values which are other schemata in the template.

The control information of whether a relational slot's value(s) has to be copied while the slot is copied is stored in the definition schema of the relational slot. A relational slot has a definition schema which has the same name as the slot, i.e. the definition schema of HAS-CAPABILITY relation is has-capability, so as the has-signal and uses-protocol. In each of the definition schemata there is a slot called "copy" which if its value is "yes" indicates that the values of this relational slot (those are the schemata linked through this relation) must be copied, otherwise the values of this relational slot has to be deleted.

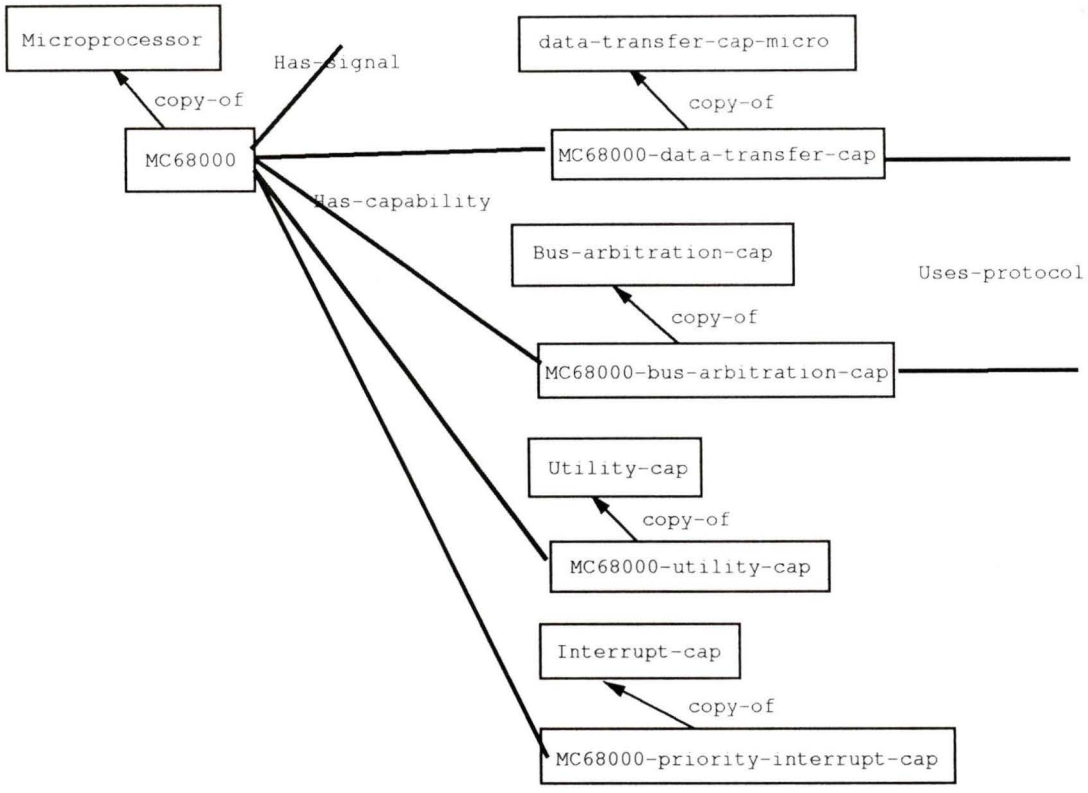

Figure 4.2: The partial structure of the MC68000 that is copied from the microprocessor template.

The copied schema is not exactly the same as the original schema not only because the values of some relational slots may change, but also because the class/subclass relation is changed. The original schema in a template is a subclass of a class, so it has an IS-A relational slot which links itself to its class schema. When the original schema is copied, the IS-A relation and its values are also copied. But the copied schema is not a subclass of the class but an element of the original schema. Therefore the IS-A relation is deleted from the copied schema, and a COPY-OF relation is added to it and the value of this relational slot is the name of the original schema.

The copy-template function starts from the root node, which is the MICROPROCESSOR schema in the above example. It first creates a new node and names it with the name of the microprocessor component to be entered, (e.g. MC68000). This new node is a copy of the microprocessor schema, and it contains all the slots and values of the microprocessor schema. Because a microprocessor schema is related to a component schema via the relation IS-A, therefore in the microprocessor schema there is an IS-A slot with a value "component". By copying all the slots and their contents in the microprocessor schema to the MC68000 schema, this IS-A relation and its value are also copied. In the classification MC68000 is related to the microprocessor schema directly but related to the component schema indirectly, therefore we delete the IS-A slot and its value in the MC68000 schema and add a COPY-OF slot with its value as "microprocessor" instead.

Next, the function checks the relational slots to determine whether their values need to be copied or deleted. For example, the value of COPY slot in the definition schema of has-capability relation is "yes",therefore the capabilities in microprocessor template are copied into MC68000, while the value of COPY slot in the definition schema of has-signal relation is "no", so the has-signal slot is left empty. Similarly when the copy function copies a capability schema, it deletes the values of uses-protocol slot in the copied schema,

because the value of the COPY slot of the definition schema of uses-protocol relation is "no".

This copy function is not only used when a new component is created, but is also applied when a protocol is chosen. Because a protocol is described by a group of action schemata, the protocol as well as the actions have to be copied to the instance of the protocol that is used in a capability of a component.

The copy function is important because it distinguishes the information that is identical in the template and all its instances from the information that is particular to a component.

The following sections explain the details of the editor commands and menus.

4.3 Editor Windows

A component consists of a group of schemata which are linked together through relations. There are two major concerns when editing a component:

1. to access each schema comprising a component efficiently through the relations, and
2. modify a schema conveniently and consistently.

The DAME editor has the following features that address the above concerns: multiple windows each displaying different information at the same time; graphical objects that represent components and allow the user to manipulate them directly; and flexible command menus.

Figure 4.3 shows the layout of the DAME Editor. The main area of the editor screen is divided into four windows that display information of the edited schema. They are: the

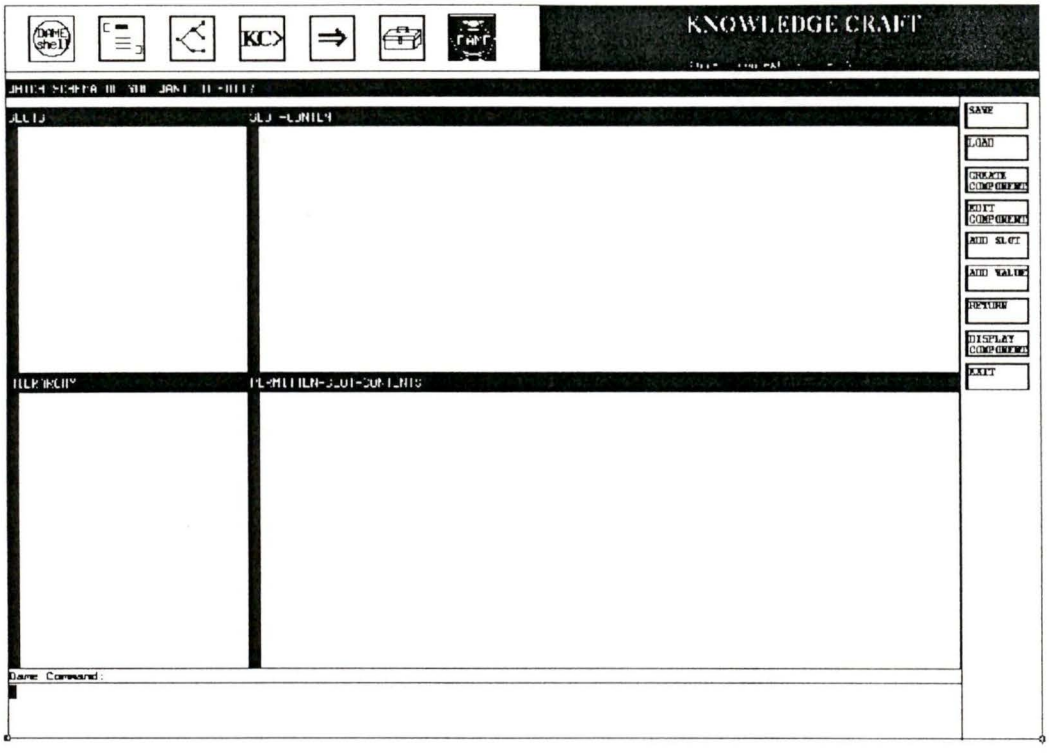

Figure 4.3: DAME Editor layout

SLOT window to display slots, the SLOT-CONTENT window to display slot values, the HIERARCHY window to display the hierarchy of the component, and the PERMITTED-SLOT-CONTENT window to display suggested slot values. At the top of these windows there is a bar which displays the current editing schema name and its type. At the left side of these windows there is a column of command icons, which are the major commands implemented for the editor. The "Dame Command" window at the bottom of the editor screen is used for user text input.

The following subsections will focus on explaining the SLOT, SLOT-CONTENT, HIERARCHY, and PERMITTED-SLOT-CONTENT windows.

4.3.1 The SLOT Window

The SLOT window is used to display the slots of the editing schema. Each slot displayed in this window can be selected by the left mouse icon. When the slot is selected, its values are displayed in the SLOT-CONTENT window, and the permitted values are displayed in the PERMITTED-SLOT-CONTENTS window.

4.3.2 The SLOT-CONTENT Window

The SLOT-CONTENT window displays the values of the selected slot. Each value in this window can be selected by either the left or the right mouse icon. When the value is selected by the right mouse button, an associated command menu is popped up. The pop-up command menu consists of three commands: EDIT VALUE, EDIT SCHEMA, and DELETE VALUE (see Figure 4.4).

The EDIT VALUE command allows the user to edit the text of the value; if the value is a schema name, the command will change the old schema name to the edited schema name.

The values of a slot which is also a relation are other schemata. When the selected value is a schema and the user wants to edit this schema, he/she can click the EDIT SCHEMA command. This command opens another editor window and displays the selected schema in the window. In this way, the user can open a series of related schemata at the same time. Figure 4.5 shows a number of editor windows which are chained together. The editor window which is invoked through the EDIT command in a previous editor is called a child editor, while the editor in which the EDIT command was invoked is called the parent editor. In the example, the editor for ACTION-1 schema is the child of the editor for HANDSHAKE-PROTOCOL schema.

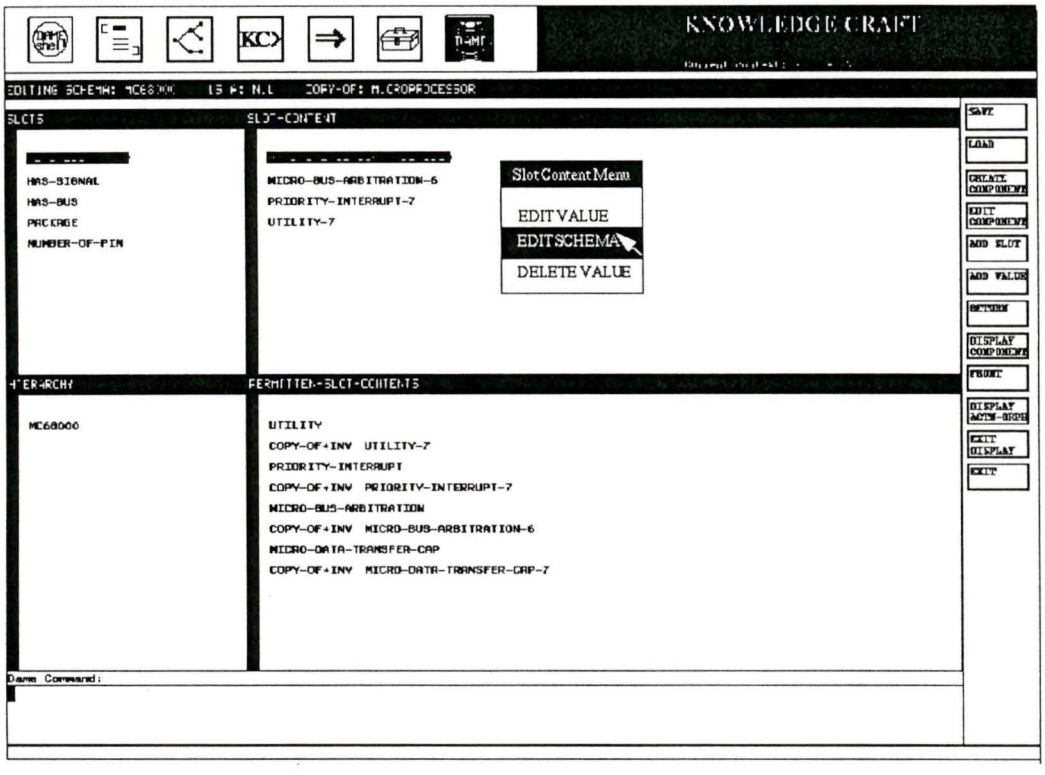

Figure 4.4: SLOT CONTENT window's pop-up-menu

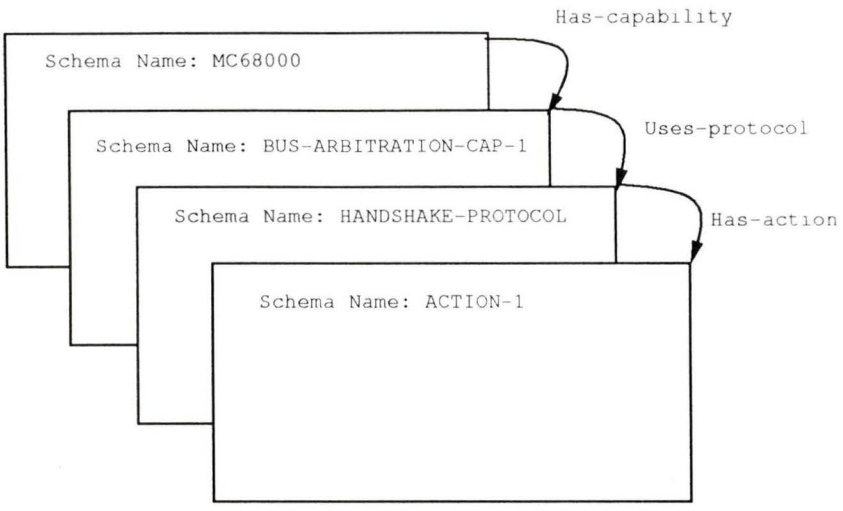

Figure 4.5: DAME Editor windows which are chained together

The DELETE VALUE command deletes the selected value from the slot.

4.3.3 The PERMITTED-SLOT-CONTENTS Window

As mentioned in section 4.2, the protocols a component capability uses have to be selected from the set of protocols listed in the template. The user is not allowed to enter values other than those included in that set because the design rules are based on that set of protocols. Other slot values which are specific to the component must be entered by the user, for example, the names of signals in the component, their associated pin numbers.

The PERMITTED-SLOT-CONTENTS window displays the suggested values for the slot which are stored in the template.

As in the SLOT-CONTENT window, each value displayed in this window can be selected by the mouse as an icon, and there is an associated pop-up menu to this icon.

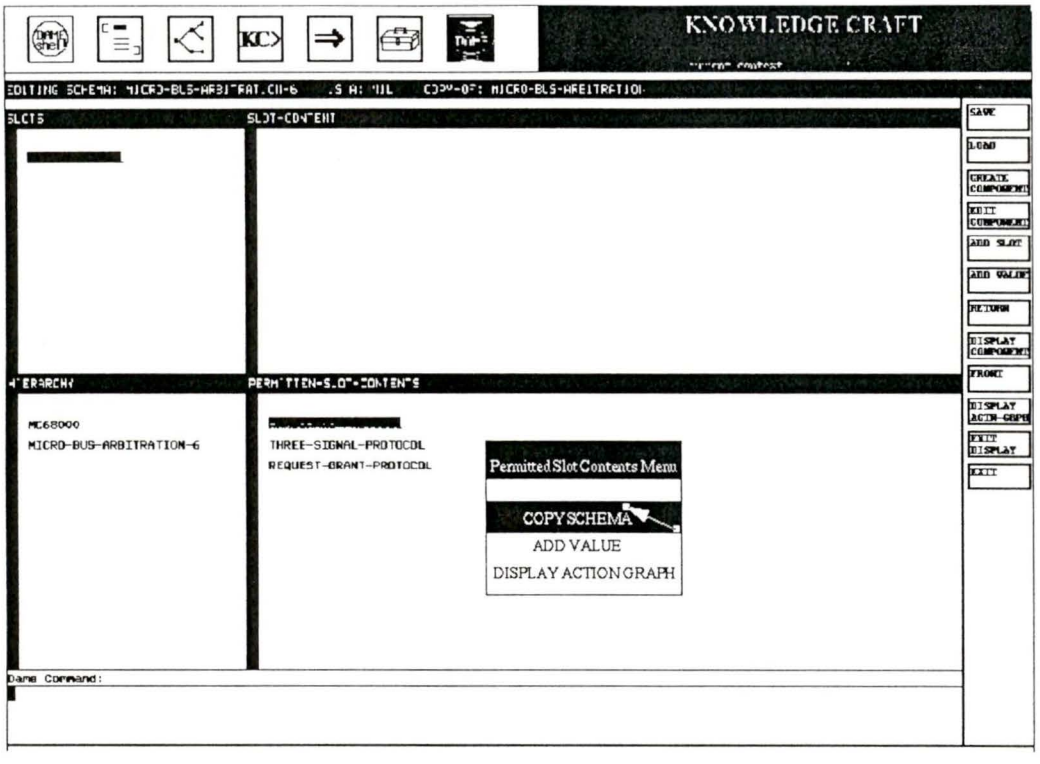

Figure 4.6: PERMITTED SLOT CONTENT window's pop-up-menu

Figure 4.6 shows the pop-up command menu for this window. The commands are: COPY SCHEMA, ADD VALUE, and DISPLAY ACTION GRAPH.

The COPY SCHEMA command allows the user to make a copy of the selected schema or a group of schemata and add this copy into the selected slot. For example, when editing the uses-protocol slot in the bus arbitration capability schema of the MC68000 component, the suggested values displayed are: HANDSHAKE-PROTOCOL, THREE-SIGNAL-PROTOCOL, and REQUEST-GRANT-PROTOCOL. If the user chooses the HANDSHAKE-PROTOCOL and clicks on the COPY SCHEMA command, a copy of the HANDSHAKE-PROTOCOL and its actions is made. The user can specify a new name

for the copy; otherwise the editor creates a default name and adds the copy in the selected slot.

The ADD VALUE command adds the selected value into the selected slot. The selected value may be a schema or just a value.

The DISPLAY ACTION GRAPH command displays the action graph of the selected protocol. If the selected value is not a protocol, this command does not have any effect. [1].

4.3.4 The HIERARCHY Window

The HIERARCHY window displays the sequence of schema-names starting from the root schema to the current editing schema. This window informs the user of the position of the current schema in the component hierarchy. For example, in Figure 4.6, the HIERARCHY window tells the user that the current editing schema is MICRO-BUS-ARBITRATION-6, which is one step down from the MC68000 root schema.

4.4 Commands of the Editor

At the right side of the editor window there are nine commands. These are the Dame Editor commands. The user can invoke a command by typing its name in the Dame Command window, or by clicking the left mouse button on the command icon. If the right mouse button is applied on a command icon, a help window is popped up with text explaining the function of the command.

The nine commands can be categorized into the following four categories:

[1] display protocol action graph function will be explained in next chapter

- commands to access the component library: SAVE and LOAD.
- commands to create a new component: CREATE COMPONENT.
- commands to edit a component: EDIT COMPONENT, ADD SLOT, ADD VALUE, DISPLAY COMPONENT, and RETURN.
- a command to exit the editor: EXIT.

The following subsections explain these commands.

4.4.1 Accessing the Component Library Commands

The component library is organized in a group of files, each storing the network of schemata describing a component. Recall that each component has a unique name which is the component file's name and also the root schema name in the file.

The SAVE command will save the component into the component library. If the component is already in the library, it will ask the user if he/she wants to overwrite the component file.

The LOAD command can load a user specified component into the working memory. It will give a pop-up message if the component does not exist in the library.

4.4.2 Create a New Component

When the CREATE COMPONENT command icon is selected, first it asks the user to enter the component name, and searches the working memory to see if a component with the same name already exists in the working memory. If a component with the same name does exist, a message is given to the user, and the command is aborted; otherwise the command

continues to request more information about the component, such as the component package name, and the number of pins. After all the information has been obtained a new component is created by copying the corresponding template and filling in the specific slots in the root schema. Then the editor enters into the edit component status, which is explained in the next subsection.

4.4.3 Editing a Component

The edit component function includes five commands: EDIT COMPONENT, ADD SLOT, ADD VALUE, DISPLAY COMPONENT, and RETURN.

The EDIT COMPONENT Command

When the EDIT COMPONENT command icon is selected, the editor asks for the component name the user wants to edit. After the component name is typed in, the editor will search for the component in the current working memory. If the component is not found in the working memory, it will search the library and will load the component into the working memory if it exists; otherwise it will tell the user the component does not exist and abort. If the component is found, the root schema name (i.e. the component name) and the type of the schema are displayed on the bar at the top of the windows, and the slots are displayed in the SLOT window.

Note that this command does not check whether the entered name is a component or not, therefore if the entered name is not a component name but a schema name, the command will also perform a retrieval. This is helpful for the DAME's system developers. It allows the developer to edit templates and other schemata of the DAME model.

The ADD SLOT Command

The ADD SLOT command can be used whenever a new slot needs to be added into the schema. After the command is selected by the left mouse button, the editor asks the user to enter the new slot name. A new slot is then created and displayed in the SLOT window.

Strictly speaking, a user is not allowed to add any slot into pre-specified schemata. This command is for the developer's using only, and it should be removed after the system's development is finished.

When the DAME system is in use, the editor should have three privileged level for different users. At level 0, it allows the developer to change the basic objects in a component model, or the structure of a template. At level 1, it allows the user to edit a component. At level 2, it only allows the user to view the component information.

The ADD VALUE Command

When a slot is selected, its values are displayed in the SLOT-CONTENT window. The ADD VALUE command can be used whenever the user wants to add a new value to this slot. The new value is entered in the Dame Command window at the bottom of the editor screen.

The RETURN Command

If the value of the selected slot is a schema, by selecting this value with the right mouse button and choosing the EDIT SCHEMA command in the pop-up menu, the user can go down one step in the component hierarchy to edit this schema. Another editor window will be opened in front of the previous one. Figure 4.5 shows that several editor windows can

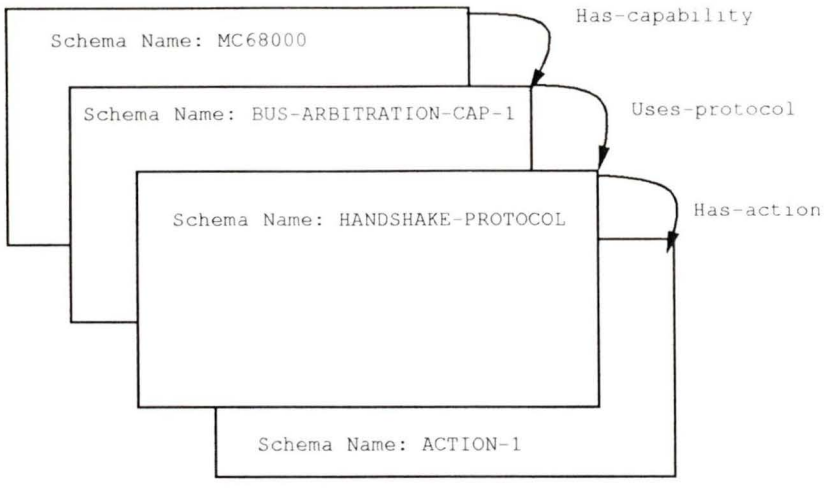

Figure 4.7: DAME Editor windows after apply a RETURN command

be opened at the same time.

The RETURN command is used to bring the parent editor window of the current one in front. Figure 4.7 shows that after applying the RETURN command in the ACTION-1 editor window, the HANDSHAKE-PROTOCOL-1 editor window is brought up in front. If the RETURN command is applied again while in the HANDSHAKE-PROTOCOL-1 editor window, the BUS-ARBITRATION-CAP-1 editor window will be brought up in front.

If the user wants to go back to the HANDSHAKE-PROTOCOL-1 editor window, he/she can select HANDSHAKE-PROTOCOL-1 from the BUS-ARBITRATION-CAP-1 window and click on the EDIT SCHEMA command again.

The DISPLAY COMPONENT Command

A component usually has many pins. The signal information of a component is pin-driven. In DAME a signal is represented by a primitive object, that is a schema with several

attributes to describe its name, pin number, I/O direction, and active state. For a user to enter and edit all the signal information of a component, and keep track of which signal has been entered and which hasn't, is very tedious. In order to ease this task for the user, the editor provides a special command: DISPLAY COMPONENT. This command can be activated only while editing a component root schema.

The function of this command is to display the component graphically in a window, as shown in Figure 4.8. Each pin in the component is a command icon. A black pin means that the corresponding signal information to that pin has already been entered, and there is a signal schema associated to this pin. The signal name is displayed beside the pin. A white pin means that the signal information has yet to be entered.

When the user clicks the left mouse button on a pin icon, a command menu will pop up. There are two commands: EDIT SIGNAL and DELETE SIGNAL. For a white pin, the EDIT SIGNAL command will ask the user to enter the signal name. The command will create a signal schema with that name and add it to the HAS-SIGNAL slot, and then open another window which displays the signal schema and allows the user to enter the detailed information such as input or output type, active state, etc. Because the pin number is already associated to the pin, the information is automatically entered to the signal schema. For a black pin, the EDIT SIGNAL command will directly open the signal schema window. The DELETE SIGNAL command deletes the signal schema from the HAS-SIGNAL slot.

The right mouse button in this window can bring up another pop-up command menu. There are three commands: BACK, RESHAPE and EXIT. The BACK command buries the component window so that the user can edit other slots. The RESHAPE command allows the user to change the size of the display window, and the EXIT command deletes the window.

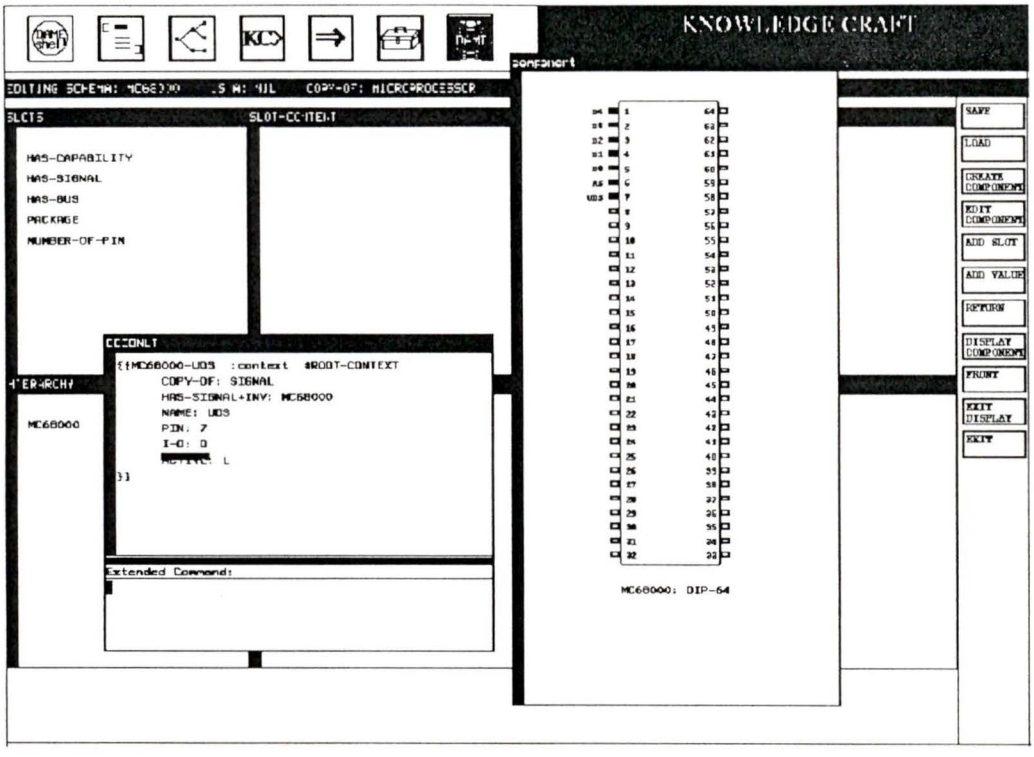

Figure 4.8: Display component window

4.4.4 The EXIT command

The EXIT command in the command icon window deletes the current editor window and its successor editors.

4.5 Summary

New microprocessor components are put on the market constantly, and so it is important to be able to incorporate the new devices as soon as they are made available into DAME's component database. In this chapter the main features of the component editor were

presented, whose main function is to acquire the new components from the user. To facilitate the acquisition procedure, a windows approach was followed, making the editor user friendly.

Firstly the copy function that manipulates portions of the semantic network to create instances using templates was explained. Then the windows structure which allows the user to navigate through the hierarchy of the component database was introduced. The windows in the editor present the schema information clearly, and the pop-up commands and command icons guide the user intuitively in performing the editing task. The DISPLAY COMPONENT command provides a convenient way for the user to enter signal information for each component. The commands related to the access of the component library are easy and safe to use. The PERMITTED-SLOT-CONTENT window displays the suggested values for the editing slot, and the values can be copied into the editing slot. If the editing slot is also a relation, the user can open another editor to edit the related schema. By using the EDIT SCHEMA in the PERMITTED-SLOT window and the RETURN command in the command icon window, the user can edit each of the schemata in a component data structure simultaneously. Finally, each command for creating and editing component instances, accessing the component library, and displaying the component graphically were discussed.

An action graph describes the dynamic behaviour of an interface protocol. To store this information, the user needs to fill in the protocol information, usually in the form of a timing diagram, into the actions of the graph. The DISPLAY-ACTION-GRAPH command in the PERMITTED-SLOT-CONTENT window's pop-up menu aids the user in recognizing the protocol actions by showing graphically the temporal relations of the action graph. Therefore it is necessary to display an action graph on a plane (the screen) which is a graph theoretical problem. In the next chapter the implementation of the DISPLAY-

ACTION-GRAPH function shall be explained.

Chapter 5

Displaying of Protocol Action Graphs

5.1 Introduction

As discussed in Section 3.2.5, the action graph of a protocol is given as the tuple (**A**,**P**,t_p). **A** is a set of actions in the protocol, **P** is the "precede" relation that guarantees the sequence of the actions in the protocol, and t_p is a function of **P** that gives the timing delay constraints between two related actions. An action graph is also a directed graph. **A** is the set of vertices, **P** is the set of edges. Figure 5.1 shows the action graph corresponding to a three signal protocol for bus arbitration.

When an action graph is displayed, it is not desirable to have a drawing with crossing edges. Therefore the problem of displaying the action graph in an editor window requires the editor to know how to draw a planar graph. In order to solve the problem, the planar algorithm [26] is used to generate a planar representation of a given action graph.

There are different planar representations as well as different types of drawings of planar

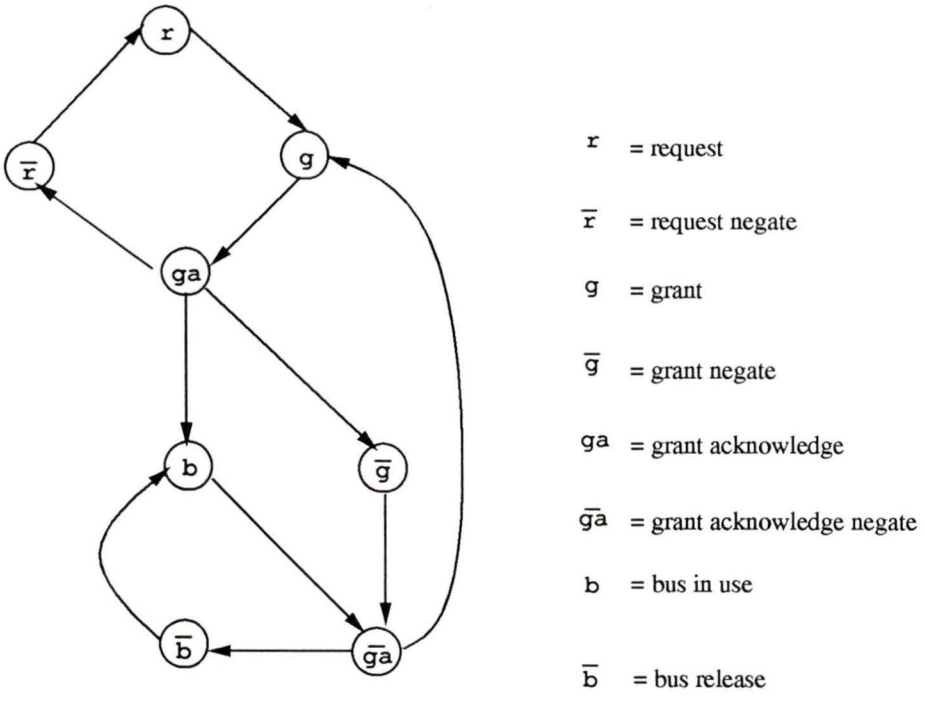

Figure 5.1: Action Graph of Bus Arbitration Three Signal Protocol

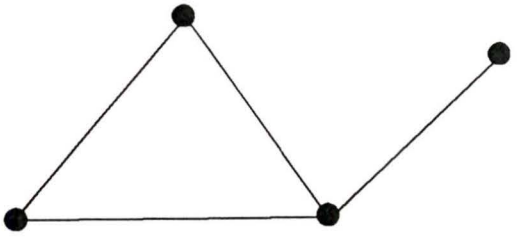

Figure 5.2: A Graph.

graphs. The basic types of drawings are: straight-line drawings [27, 28], and orthogonal grid drawings [29]. The latter one is widely used in circuit layout. In the straight-line drawings the arcs are straight lines. The DAME editor follows the straight-line drawings to draw the action graph, but not very strictly.

This chapter explains the algorithm and its implementation in the DAME editor.

5.2 Definitions

The Planar algorithm tests the planarity of a graph. Before we introduce the algorithm, we present a brief definitions of the sequence of terms used in this algorithm [30, 31, 32, 26].

- A *graph* G is defined as G = (V(G), E(G)), V(G) is a finite non-empty set of elements called *vertices*, and E(G) is a finite set of distinct unordered pairs of distinct elements of V(G) called *edges*, (see Figure 5.2).

  Let $\mathbf{G} = (V, E)$, V: $\{v_i \mid i = 0, 1, ...k\}$, $E \subseteq VXV = \{(v, w) \mid v, w \in V\}$. An edge is denoted by $h = (v, w)$, the vertices *v, w* are called the edge's *ends* or *endpoints*. For $v \neq w$, the edge is called a *link*, and the two vertices said to be *adjacent* or *neighbors*. If *v=w*, the edge is called a loop.

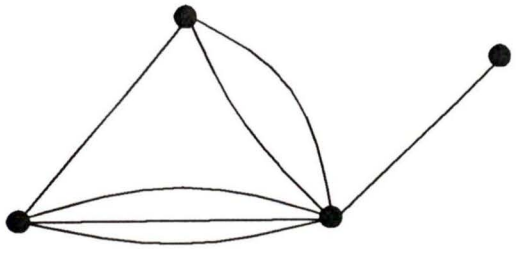

Figure 5.3: A Multigraph.

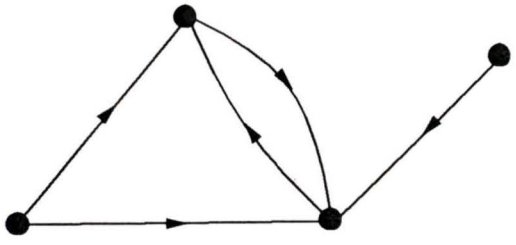

Figure 5.4: A Directed Graph.

If E is not a distinct set of pairs, that is two or more edges joining the same pair of vertices, then the graph is called a multigraph (see Figure 5.3).

- A *directed graph* or *digraph* is defined as **D**=(V(**D**), A(**D**)), where V(**D**) is a finite non-empty set of elements called vertices, and A(**D**) is a finite set of distinct ordered pairs of distinct elements of V(**D**) called *arcs* (see Figure 5.4).

- A *subgraph* of a graph **G**=(V(**G**), E(**G**)) is a graph **H**=(V(**H**), E(**H**)) such that $V(\mathbf{H}) \subseteq V(\mathbf{G})$ and $E(\mathbf{H}) \subseteq E(\mathbf{G})$.

- For a subgraph **H** of a graph **G**, a *vertex of attachment* of **H** in **G** is a vertex of **H** that is incident in **G** with an edge not belonging to E(**H**). The set of vertices of attachment of **H** in **G** is denoted by W(**G**, **H**).

- A *sequence* in a graph **G** is defined as a series of vertices $v_0, v_1, \ldots, v_k$ such that $\{v_{i-1}, v_i\}$ is an edge for each $i = 1, \ldots, k$. If the vertices are pairwise distinct, the sequence is called a *path*. If $v_0 = v_k$, and $v_i \neq v_j$ for all $1 = i < j = k$, the path is called a *circuit* or a *cycle*.

- A graph is *connected* if there is a path joining each pair of vertices of **G**; a graph which is not connected is called *disconnected.*

  The graphs we use in the rest of the chapter are defined as finite, connected, loopfree graphs.

- Let **H**=(V(**H**), E(**H**)) be a subgraph of graph **G**=(V(**G**), E(**G**)), for all edges $h=\{v, w\}$, if $v, w \in V(\mathbf{H})$, and $h \in E(\mathbf{H})$, then **H** is called a maximal connected subgraph of **G**, or a *component*. The endpoints of a component is a set of vertices of attachment W(**H**, **G**).

- If **H**=(V(**H**), E(**H**)) is a subgraph of **G**=(V(**G**), E(**G**)), V(**H**) is the set of *interior vertices*, $V(\mathbf{G}) - V(\mathbf{H})$ is the set of *exterior vertices.*

- A *chord* of a subgraph **H** of graph **G** is a path in **G** whose endpoints are on **H** and the rest of the points and edges are not on **H**. If a chord has only two vertices, it is called a *simple chord.* If a chord whose endpoints are the same, it is called *degenerate.*

- An *exterior component* of a subgraph **H** in a graph **G** is a maximal connected subgraph $\mathbf{H}' = (V(\mathbf{H})', E(\mathbf{H}'))$ of **G** if any two of the vertices in V(**H**′) lie on a chord of **H**. The set of vertices of attachment of **H**′ to **H** is $W(\mathbf{H}', \mathbf{H}) = V(\mathbf{H}) \cap V(\mathbf{H}')$.

- The *drawing* of a graph **G** is to assign each vertex a point in the plane, and to each edge a connected simple curve connecting its ends.

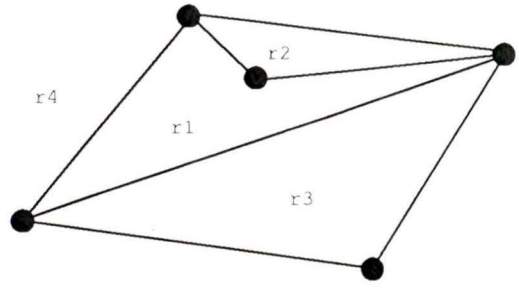

Figure 5.5: A plane graph.

- A *planar graph* is a graph which can be drawn on the plane in such a way that no two edges intersect geometrically except at a vertex to which they are both incident. A graph embedded on the plane in this way is called a *plane graph.*

- In a plane graph **G**, the regions of the plane not on **G** are partitioned into open sets called *faces* (see Figure 5.5, it has four faces: *r1, r2, r3, r4*). The cycle going around a face is called a *mesh.* A mesh is denoted by the cycle $(v_0, v_1, ..., v_k, v_0)$ surrounding the face such that when the cycle is traversed the surrounded face is always on the right-hand side.

Figure 5.6 gives an example of graph, subgraph, exterior vertices, exterior components, and chords. Give a graph $\mathbf{G} = (V(\mathbf{G}), E(\mathbf{G}))$ which has 9 vertices and 17 edges:

$V(\mathbf{G}) = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$,

$E(\mathbf{G}) = \{(1,2), (1,5), (1,6), (1,7), (1,8), (2,3), (2,7), (2,8), (2,9), (3,4), (3,7), (3,9), (4,5), (4,8), (4,9), (5,6), (8,9)\}$;

$\mathbf{H} = (V(\mathbf{H}), E(\mathbf{H}))$ is a subgraph of **G**:

$V(\mathbf{H} = \{1, 2, 3, 4, 5, 6\}$,

$E(\mathbf{H}) = \{(1,2), (2,3), (3,4), (4,5), (5,6)\}$;

H is also a cycle: $\{1, 2, 3, 4, 5, 6, 1\}$; the set of vertices $\{7, 8, 9\}$ are the exterior vertices of

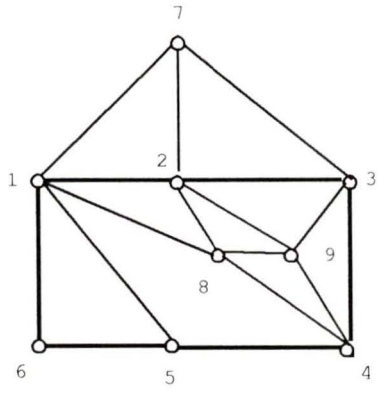

Figure 5.6: An example of graph, subgraph, exterior vertices, exterior components, and chords.

H; **H** has three exterior components: $\{1,2,3,7\}$, $\{1,2,3,4,8,9\}$, and $\{1,5\}$; (1,8,4), (1,5), and (2,8,9,2) are chords of **H**, (1,5) is a simple chord, (2,8,9,2) is a degenerate chord.

5.3 The Planar Algorithm

This algorithm, proposed by Rubin [26], tests the planarity of a given finite, connected, undirected, loopfree graph, and at the same time it generates the planar representation of the given graph if it is planar.

The basic idea of the algorithm is to choose a connected planar subgraph **H** of the given graph **G**, and find **H**'s exterior components in **G**. If there is one exterior component that cannot fit into any of the faces of **H**, then this graph **G** is not a planar graph and the algorithm ends. Otherwise choose one chord from one of the components, embed the chord into the face it belongs, divide the face into two faces, add the chord to the subgraph, $\mathbf{H} = \mathbf{H} +$ the chord . Repeat the algorithm for the updated subgraph. If no exterior

component is found then the graph G is proven a planar graph.

The planar algorithm consists of the following steps:

Step 1 Select a cycle as the initial subgraph. The initial unexplored vertices are the vertices of attachment of the subgraph to the given graph. The initial subgraph divides the plane into two faces: the region inside the cycle and the region outside the cycle which are denoted by two meshes.

Step 2 Since all exterior components of the subgraph can be embedded in either of the meshes, choose a chord and embed it in the mesh which surrounds the region inside the cycle. Then split the mesh into two by the chord and add the chord into the subgraph. Update the unexplored vertices list of the subgraph.

Step 3 Explore the unexplored vertices on the subgraph, find exterior components of the subgraph. If not found, inform that the graph is planar and stop.

Step 4 If one exterior component is found that can not be embedded in any mesh, then the graph is non-planar: stop.

Step 5 If one exterior component of the subgraph is found that can be embedded in one and only one mesh, then choose a chord from the exterior component and embed it in the mesh, split the mesh into two by the chord, update the unexplored vertices list of the subgraph, and go to step 3.

Step 6 If all exterior components of the subgraph can be embedded in more than one mesh, choose a chord from one component and embed it in one mesh, and split the mesh to two new meshes by the chord, update the unexplored vertices list of the subgraph. Go to step 3.

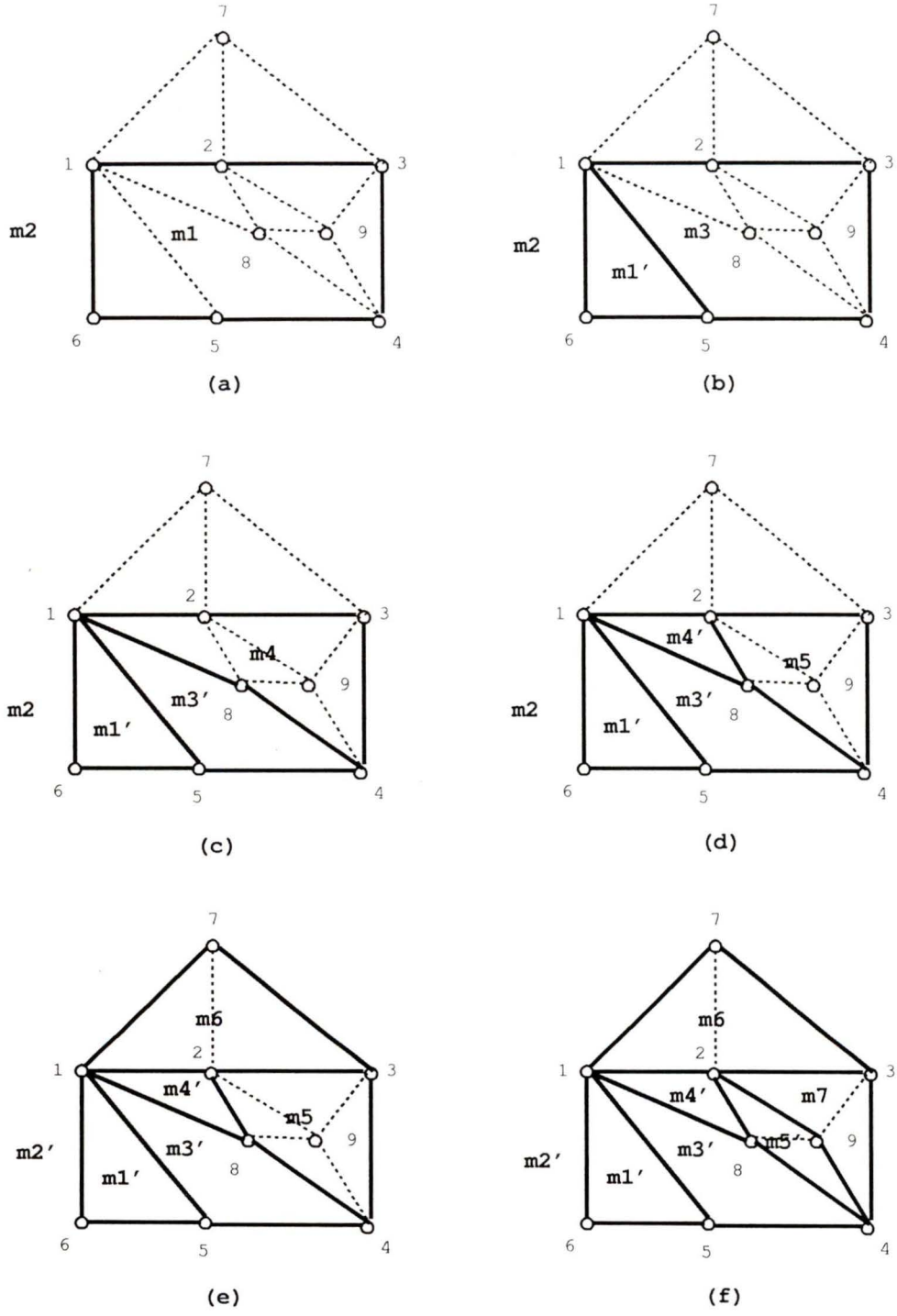

Figure 5.7: An Example of the Algorithm

We use the graph **G** shown in Figure 5.6 to explain how the planar algorithm works.

(a) First, select the initial cycle (1,2,3,4,5,6) as the connected subgraph **H** of **G**: $V(\mathbf{H}) = \{1, 2, 3, 4, 5, 6\}$, $E(\mathbf{H}) = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1)\}$. The unexplored vertices list is (1,2,3,4,5), the two initial meshes are m1 = (1,2,3,4,5,6,1) for the face inside the initial cycle, and m2 = (1,6,5,4,3,2,1) for the rest of the plane (see (a) of Figure 5.7).

(b) There are three exterior components of the subgraph: {1,5}, {1,2,7,3}, and {1,2,3,4,8,9}. Choose chord (1,5), place it in mesh m1, and the new meshes are m1' = (1,5,6,1), and m3 = (1,2,3,4,5,1). The unexplored vertices list is (1,2,3,4). The subgraph is now $V(\mathbf{H}) = \{1, 2, 3, 4, 5, 6\}$, and $E(\mathbf{H}) = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1), (1, 5)\}$ (see (b) of Figure 5.7).

(c) The above subgraph has two exterior components: {1,2,3,4,8,9}, and {1,2,3,7}. Both of the components can be placed in either m2 or m3. Choose a chord (1,8,4) from the component {1,2,3,4,8,9}, place it on mesh m3, the new meshes are $m3' = (1, 8, 4, 5, 1)$ and $m4 = (1, 2, 3, 4, 8, 1)$. The list of the unexplored vertices is (1,2,3,4,8) (see (c) of Figure 5.7). The subgraph is $V(\mathbf{H}) = \{1, 2, 3, 4, 5, 6, 8\}$, and $E(\mathbf{H}) = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1), (1, 5), (1, 8), (8, 4)\}$.

(d) The exterior components of the subgraph are: {1,7,3,2} and {2,3,4,8,9}. The latter one only can be embedded in mesh m4. Choose a chord (2,8) from the component {2,3,4,8,9}, split m4 into two new ones, $m4' = (1, 2, 8, 1)$ and $m5 = (2, 3, 4, 8, 2)$. The list of the unexplored vertices is still (1,2,3,4,8) (see (d) of Figure 5.7). The subgraph is $V(\mathbf{H}) = \{1, 2, 3, 4, 5, 6, 8\}$, and $E(\mathbf{H}) = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1), (1, 5), (1, 8), (2, 8), (8, 4)\}$.

(e) Find exterior component {1,7,3,2}; it can be embedded in mesh m2 only. Choose

chord (1,7,2), split m2 into two new ones, $m2' = (1, 6, 5, 4, 3, 7, 1)$ and $m6 = (1, 7, 3, 2, 1)$. The unexplored vertices list is (2,3,4,8,7) (see (e) of Figure 5.7). The subgraph is $V(\mathbf{H}) = \{1, 2, 3, 4, 5, 6, 8, 7\}$, and $E(\mathbf{H}) = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1), (1, 5), (1, 8), (2, 8), (8, 4), (1, 7), (7, 2)\}$.

(f) Find exterior component {2,8,4,3,9}, and it can be embedded in mesh m5 only. Choose chord (2,9,4), split m5 into $m5' = (8, 2, 9, 4, 8)$ and $m7 = (2, 3, 4, 9, 2)$. The unexplored vertices list is (2,3,8,7,9) (see (f) of Figure 5.7). The subgraph is $V(\mathbf{H}) = \{1, 2, 3, 4, 5, 6, 8, 7, 9\}$, and $E(\mathbf{H}) = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1), (1, 5), (1, 8), (2, 8), (8, 4), (1, 7), (7, 2), (2, 9), (9, 4)\}$.

By continuing this algorithm, the simple chord (2,7), (8,9) and (3,9) are found and added into the subgraph. The algorithm ends and the graph is a planar graph.

5.4 Applying the Planar Algorithm on Action Graphs

In DAME, action graphs are directed graphs. The original action graphs can be obtained from the DAME database as follows. *Get-values* is a function which returns a list of slot values of a schema. By applying this function on a slot of a schema, that is (*get-values a-protocol-schema* HAS-ACTIONS), the set of nodes (vertices) of the action graph of the protocol can be obtained. Each node itself is a schema called action schema. Through the PRECEDES relation in an action schema, the list of nodes that succeed an action can be obtained, that is (*get-values an-action-schema* PRECEDES).

In an action graph one of the vertices is considered significant in the sense that it signifies the start of an information exchange transaction. Thus the longest cycle through this vertex usually corresponds to the control part of the protocol. This significant vertex is called the

initial vertex of the action graph. When an action graph is drawn on a plane, we want to place the initial vertex at the top and draw the longest cycle first.

In order to be able to use the planar algorithm, the action graph is converted into an undirected graph. The display action graph routine has the following steps:

1. Get an action graph **G** from a protocol schemata, and represent it in a list.
2. Find the initial cycle of **G** which is the longest cycle among all the cycles that pass through the given vertex, and let it be the subgraph **H** of **G**.
3. Convert **G** into the corresponding undirected graph $\mathbf{G}'$, and **H** into the undirected graph $\mathbf{H}'$.
4. Apply the planar algorithm on graph $\mathbf{G}'$ and subgraph $\mathbf{H}'$, the return of the algorithm is a set of lists of meshes, $\mathbf{L} = (m_1, m_2, ..., m_s)$, if the graph is planar, otherwise is $\mathbf{L} = nil$.
5. Transfer the set of meshes into a matrix which places each node in a relative position on a plane.
6. Display the nodes on a screen window according to the matrix, display the set of arcs according to the original graph **G**.

5.5 Representation of Graphs

In DAME, we represent graphs as lists of lists. Each second-level list consists of a node followed by all its neighbors.

Let $\{n_i\}, i = 1, 2, ..., k$ be the set of nodes in a graph, $N(n_i)$ be a list of nodes which are the neighbors of n_i. An undirected graph can be fully represented by the list:

$((n_1\ N(n_1))\ (n_2\ N(n_2))\ ...(n_k\ N(n_k)))$.

Let nb_j be a node in $N(n_i)$, $(n_i\ nb_j)$ represents an edge which n_i is connected to. An undirected edge connecting nodes *u* and *v* occurs twice in this description: once because *u* is in $N(v)$, and once because *v* is in $N(u)$. This representation of graphs is called *adjacency list.*

This representation can also describe directed graphs. In a directed graph, an arc has an initial vertex and a terminal vertex. Let $\{n_i\}, i = 1, 2, ..., k$ be the set of nodes in a directed graph, $S(n_i)$ be a list of nodes which are the terminal vertices of arcs whose initial vertex is n_i. A directed graph is described by the list:

$((n_1\ S(n_1))\ (n_2\ S(n_2))\ ...(n_k\ S(n_k)))$.

In this description, an arc from node *u* to node *v* only occurs once and is described by the occurrence of *v* in $S(u)$.

Let $((n_1\ S(n_1))\ (n_2\ S(n_2))\ ...(n_k\ S(n_k)))$ be an original action graph, and $((n_1\ N(n_1))\ (n_2\ N(n_2))\ ...(n_k\ N(n_k)))$ be the converted undirected graph.
$N(n_i) = S(n_i) + P(n_i)$, where $P(n_i)$ is the set of nodes precede n_i. $P(n_i)$ can be obtained from the successor-list of other nodes, $S(n_j), j = 1, 2, ..., k, j \neq i$. The set of precedent nodes of n_i can be found in the following way:

a node $n_j \in P(n_i)$ *iff* $n_i \in S(n_j), j = 1, 2, ...k, j \neq i$.

5.5.1 Exterior Components

After the subgraph is initialized with the initial cycle from the original action graph and the action graph **G** and the subgraph **H** converted to undirected graphs $\mathbf{G}'$ and $\mathbf{H}'$, the planar algorithm can be applied.

Then the undirected graph $\mathbf{G}'$ is:

$\mathbf{G}' = ((n_1\ N(n_1))\ (n_2\ N(n_2))\ ...(n_k\ N(n_k)))$,

and the subgraph $\mathbf{H}'$ is:

$\mathbf{H}' = ((v_1\ N'(v_1))\ (v_2\ N'(v_2))\ ...(v_l\ N'(v_l)))$,

where $N'(v_j)$ is a list of neighbors of vertex v_j in the subgraph $\mathbf{H}'$.

One of the key functions of the algorithm is to find the exterior components of the subgraph. The search starts from the unexplored vertices list. The unexplored neighbors of a vertex on the subgraph is the vertex's neighbors on the graph $\mathbf{G}'$ but not on $\mathbf{H}'$. Let U be the unexplored vertices list: $U = ((v_1\ N''(v_1))\ (v_2\ N''(v_2))\ ...(v_l\ N''(v_l)))$, where $N''(v_j) = N(n_i) - N'(v_j)$ if $n_i = v_j$. If the set of unexplored neighbors of vertex v_j is empty, that is this vertex has no unexplored neighbors, then $(v_j\ N''(v_j))$ is deleted from the list.

To find an exterior component of the subgraph we check the unexplored neighbors list. If one of the unexplored neighbors of a vertex is on the subgraph, then it is an exterior component consisting only of a simple chord. For example in Figure 5.6 (b), neighbor 5 is explored from vertex 1. If a neighbor of the vertex is not on the subgraph, then it is an exterior vertex. Explore all the neighbors of the exterior vertex; they are either the endpoints of the exterior component (that is they are on the subgraph), or the new exterior vertices of the subgraph (that is they are not on the subgraph). Continue to explore the new

exterior vertices until all the endpoints of the component are found.

For example, in Figure 5.6 (c), vertex 8 is explored from vertex 1 as an exterior vertex. Explore the neighbors of vertex 8 (2, 9, 4). 2 and 4 are the endpoints of the exterior component; put them in an endpoints list (1 2 4). Put 9 in an unexplored exterior vertex list (9). Continue to explore the unexplored exterior vertices; the neighbors are 2, 3, and 4, they are all endpoints. By now the updated endpoint list is (1 2 3 4), and the unexplored exterior vertices list is empty; therefore the exterior component has been found.

When an exterior component is found, the algorithm tries to embed the component into the meshes. A mesh is represented as an ordered list of vertices. If the endpoints of an exterior component belong to a mesh list, then this exterior component can be embedded in this mesh. If a component cannot be embedded into any mesh, that is the endpoints of the component are not a subset of any mesh list, the graph is not planar. If it can fit into only one mesh, then one chord is chosen from the component and added into the subgraph, the search of exterior components of the updated subgraph will start again. If the component can fit into more than one mesh, this component will be put into a component list, and when all the exterior components are found and all of them can be embedded into more than one mesh, a chord of the first component in the list is chosen and embedded into one of the meshes.

5.5.2 Transform the Mesh Lists into a Matrix

The return value from the execution of the planar algorithm is a list of meshes. For example, in Figure 5.1, the mesh lists is (a1 a2 a3 a4 a1), (a2 a6 a8 a7 a3 a2), (a8 a6 a7 a8), (a6 a5 a3 a7 a6), (a1 a4 a3 a5 a6 a2 a1).

When the graph is drawn, the shape of the graph is dependent on which mesh is the one

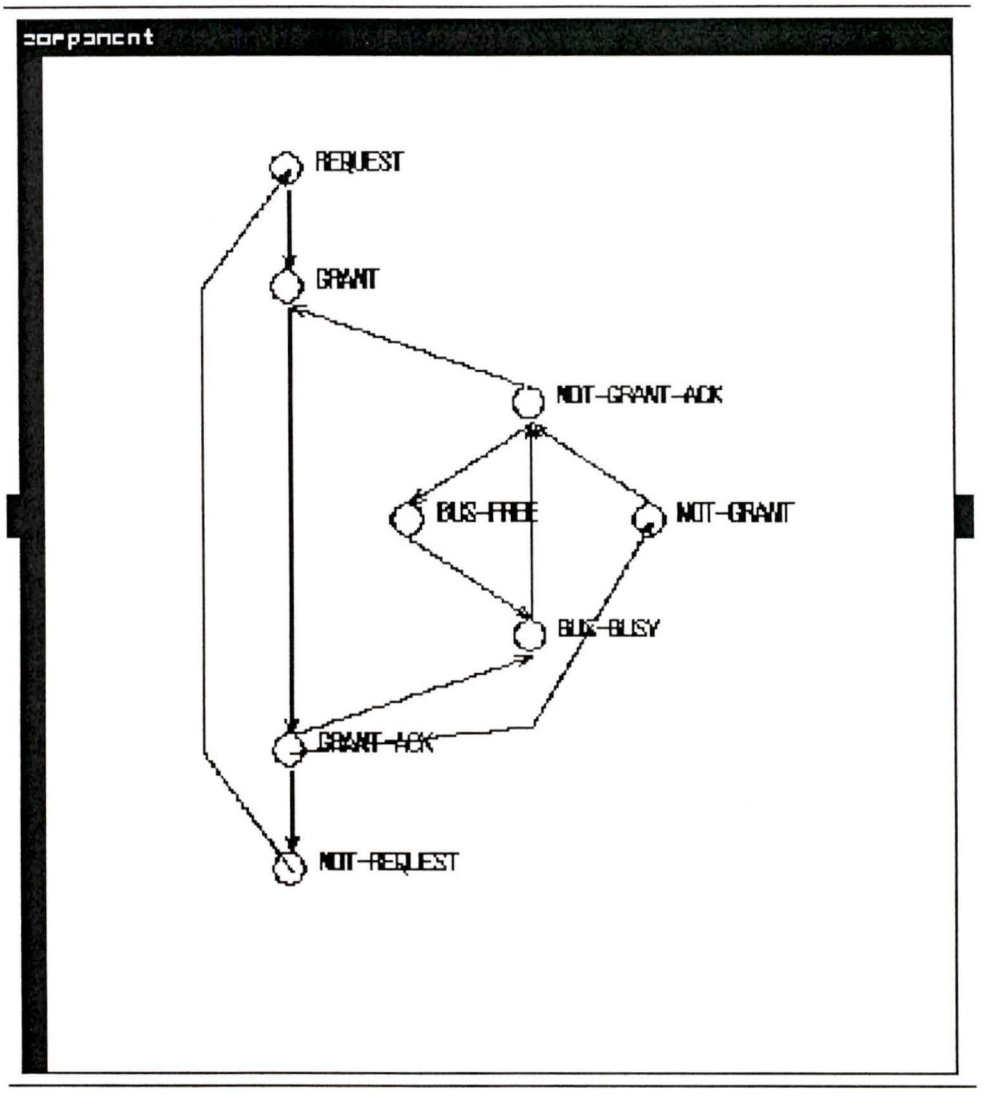

Figure 5.8: An Example of Display Action Graph

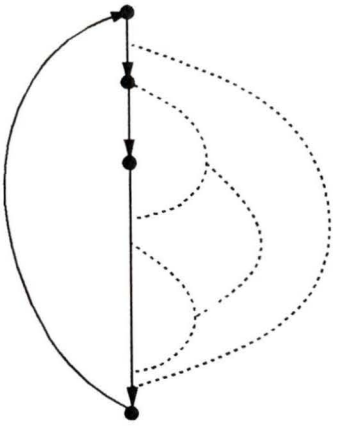

Figure 5.9: Drawing an Action Graph

drawn first. In our case we want to draw the initial vertex at the top of the drawing and draw the initial cycle first. Therefore we find the mesh which matches the initial cycle, place it in a column of the matrix and keep the same order as in the mesh list and the initial vertex at the top of the column. Next we place the rest of the meshes in the matrix. Figure 5.9 shows the way we draw the meshes. Because we place the vertices of the first mesh on a vertical line, the rest of the meshes will be placed on the right side of the first mesh.

In order to choose the next mesh to be placed beside the one already drawn, we defined the boundary as the set of ordered vertices which when you walk through they circle the drawn partial graph inside. Compare the set of vertices on a mesh and the set of vertices on the boundary, there are some meshes that have no joint vertex on the boundary, other meshes have one or more vertices joint with the boundary. There may be more than one mesh adjacent to the boundary. The mesh which has the most vertices joint with the boundary is chosen. If there are more than one such meshes, choose the one at the top. The chosen mesh has part of the vertices already on the boundary, and the rest of the vertices are needed to add into the matrix. We add a new column, column l_{i+1}, into the matrix and

place these vertices in the column. There are two vertices on the boundary which link to the other part of the mesh on column l_{i+1}, denoted as v_t and v_b, and they are on row r_a and r_b respectively. The vertices on column l_{i+1} are places between row r_a and r_b. If there are not enough rows between r_a and r_b for placing the vertices, then insert some new rows to the matrix. After a mesh is placed, update the boundary. Repeat the procedure until the boundary matches the last mesh, that is the mesh that circles the rest of the plane.

In the example of Figure 5.1, the list of meshes are (a1 a2 a3 a4 a1), (a2 a6 a8 a7 a3 a2), (a8 a6 a7 a8), (a6 a5 a3 a7 a6), (a1 a4 a3 a5 a6 a2 a1), and the initial cycle is (a1 a2 a3 a4 a1). Place this cycle in a matrix:

$a1$

$a2$

$a3$

$a4$

First consider that this cycle is the boundary, and choose a mesh which is the most vertices on the current boundary. In this case (a2 a6 a8 a7 a3 a2) is chosen. Because the order of the sequence of a mesh uses the right hand side rule, the area that belongs to the mesh is always on the right hand side. Therefore nodes a6, a7 and a8 are at the right and between nodes a2 and a3. The resulting matrix is:

$a1$ 0

$a2$ 0

0 $a6$

0 $a8$

0 $a7$

$a3$ 0

$a4$ 0

The boundary now is (a1 a2 a6 a8 a7 a3 a4 a1). The next adjacent mesh to the boundary is (a8 a6 a7 a8). These three nodes are already on the current boundary, but the updated boundary is (a1 a2 a6 a7 a3 a4 a1). In order to draw the link between a6 and a7 without overlapping the links between a8 a6 and a8 a7, we need to adjust the matrix to:

$a1$ 0 0

$a2$ 0 0

0 0 $a6$

0 $a8$ 0

0 0 $a7$

$a3$ 0 0

$a4$ 0 0

The next adjacent mesh is (a6 a5 a3 a7 a6). Add node a5 at the right of the boundary, and update the boundary to (a1 a2 a6 a5 a3 a4 a1). This is the same set of nodes as in the last mesh determined at the beginning, (a1 a4 a3 a5 a6 a2 a1), but the nodes appear in the opposite order. Therefore all the meshes are now placed in the matrix. The final matrix is:

$a1$	0	0	0
$a2$	0	0	0
0	0	$a6$	0
0	$a8$	0	$a5$
0	0	$a7$	0
$a3$	0	0	0
$a4$	0	0	0

According to this position matrix of the action graph and the size of a displaying window, the coordinates of each vertex can be calculated, and the vertices are drawn in the window. The arcs of the graph can be obtained from the original action graph and displayed in the window. Figure 5.8 shows the display of the three signal protocol action graph.

5.6 Summary

This chapter first introduces some of the terminology of graph theory. The planar algorithm and how it is applied on action graphs are then explained. The main idea of the algorithm is to test the planarity of a graph. First a connected planar subgraph is initialized from a given graph; find the exterior components of the subgraph; if there is one component that cannot be embedded into any of the faces of the planar subgraph, the algorithm is terminated and this given graph is not a planar graph; otherwise choose a chord from one component and embed it to one of the faces of the subgraph which divides the face into two faces. Repeat the procedure until either one component which cannot be embedded into any faces is found or the given graph is proven planar.

In order to apply this algorithm on action graphs, we choose the initial vertex of an action graph and its longest cycle as the initial subgraph, convert both the action graph and

the subgraph into undirected graphs, and then apply the algorithm to them.

Also, the list representation of directed and undirected graphs is introduced, and the function of search exterior components is explained. Finally, we discussed how to draw the action graph on a display window. We transform the list of meshes into a matrix in which the vertices are placed in the relative positions.

Chapter 6

Conclusions

The purpose of this work is to address the capture of knowledge in DAME, an expert system for automated design of microprocessor-based systems. Due to technological advances, the knowledge base of such a system has to be constantly updated, both with new component information and with new design rules.

In order to cope with the addition of design knowledge, DAME models components at the protocol level. Protocols are abstractions of the inter-component information transfer. In DAME a few general rules produce a design which is dependent only on the interfacing protocols used by the components in the system and not on particulars such as signal names and polarities. The design knowledge growth is controlled because new protocols and capabilities do not appear as frequently as new components do.

The DAME editor is a specialized user interface that assists the user with the incorporation of new devices into the component database. This work describes DAME's editor whose responsibility is to assist the user in adding the component information the DAME designer requires to produce a working design.

DAME's component model structures the component information into a frame hierarchy which efficiently describes the component database by organizing the information into several abstraction levels that have a direct correspondence to the levels of the design.

Components are classified into classes according to their functions. Each class has a template that characterizes the behaviour of the components in the class.

Based on the component model and component templates, the DAME editor provides commands for creating new components, accessing the component library, and editing components. One of the basic functions of the editor is the copy template function which copies a partial template structure to the new component. Thus the user has to fill in only component dependent information, such as component names, signals, etc.

Editing component signals is a tedious job, because each component may have as many as hundreds of signals with different signal names, pin numbers, directions, etc. In order to make this job easier, the DAME editor provides a special window which displays the editing component in a graphic form. With the signal names attached to each pin, the user can know how many signals have been added.

Protocols are stored in DAME in the form of action graphs. Nodes in the graph correspond to the elementary operations in the protocol sequence also called actions, and edges describe a precedence relation between the actions. The DAME editor also provides a tool that draws the action graph of a protocol in a planar form whenever possible.

The DAME editor is part of DAME's user interface. It undertakes the knowledge acquisition task. At this development stage, we include some extra functions in the editor, allowing the developer to edit any schema, not just the schemata in the component library. These functions have to be deleted from the editor after the system is developed.

Bibliography

[1] D. A. Waterman and F. Hayes-Roth, *Pattern-Directed Inference Systems*. Academic Press, 1978.

[2] D. S. Prerau, *Developing and Managing Expert Systems*. Addison-Wesley, 1990.

[3] A. Gupta and B. E. Prasad, *Microcomputer-Based Expert Systems*. IEEE Press, 1988.

[4] S. L. Tanimoto, *The Elements of Artificial Intelligence*. Computer Science Press, 1990.

[5] P. Jackson, *Introduction to Expert System*. Addison-wesley Publishing Company, 1990.

[6] M. Stelzner and M. D. Williams, *The Evolution of Interface Requirements for Expert Systems*, ch. 12. Expert Systems: the User Interface, Ablex Publishing Corporation, 1988.

[7] D. Nau and M. Gray, *Hierachical Knowledge Clustering: A Way to Represent and Use Problem-solving Knowledge*, ch. 5. Expert Systems: the User Interface, Ablex Publishing Corporation, 1988.

[8] G. S. Tuthill, *Knowledge Engineering: Concepts and Practices for Knowledge-Based Systems*. TAB Books Inc., 1990.

[9] M. D. S. Tuhrim, *Expert System Development: Letting the Domain Specialist Directly Author Knowledge Bases*, ch. 3, pp. 37–55. Expert Systems: the User Interface, Ablex Publishing Corporation, 1988.

[10] J. A. Hendler, *Expert Systems: the User Interface*. Ablex Publishing Corporation, 1988.

[11] J. McDermott, "R1: A rule-based configurer of computer systems," Tech. Rep. CMU-CS-80-119, Carnegie Mellon University, 1980.

[12] P. S. Rosenbloom, J. E. Laird, J. McDermott, A. Newell, and E. Orciuch, "R1-soar: an experiment in knowledge-intensive programming in a problem-solving architecture," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. pam1-7, Sept. 1985.

[13] A. van de Brug, J. Bachant, and J. McDermott, "Doing r1 with style," in *Proc. 21th ACM/IEEE Design Automation Conference*, 1985.

[14] J. Bachant and J. McDermott, "R1 revisited: Four years in the trenches," *The AI Magazine*, fall 1984.

[15] M. Smith and J. Bowen, "Knowledge and experience-based systems for analysis and design of microprocessor," *Microprocessors and Microsystems*, vol. 6, pp. 515–518, Dec. 1982.

[16] R. Barletta, "An introduction to case-based reasoning," *AI EXPERT, the Magazing of Artificial Intelligence in Practice*, Aug. 1991.

[17] W. P. Birmingham and D. P. Siewiorek, "Micon: A knowledge based single board computer designer," in *Proc. 21st Design Automation Conference*, 1984.

[18] W. P. Birmingham, A. P. Gupta, and D. P. Siewiorek, "The micon system for computer design," *IEEE Micro*, Oct. 1989.

[19] W. P. Birmingham, A. Kapoor, D. P. Siewiorek, and N. Vidovic, "The design of an integrated engironment for the automated synthesis of small computer systems," in *Proc. 26th ACM/IEEE Design Automation Conference*, 1989.

[20] M. D. Rychener, *Expert Systems for Engineering Design*. Academic Press Inc., 1988.

[21] Y.-H. Kuo, L.-Y. Kung, C.-C. Tzeng, G.-H. Jeng, and W.-K. Chia, "Kmds: An expert system for integrated hardware/software design of microprocessor-based digital systems," *IEEE Micro*, 1991.

[22] N. Dimopoulos, B. Huber, K. Li, D. Caughey, M. Escalante, D. Li, R. Burnett, and E. Manning, "Modelling components in dame," in *Proceedings of the 3rd International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, (Charleston, South Carolina), pp. 716–725, July 1990.

[23] B. Huber, K. Li, N. Dimopoulos, D. Li, R. Burnett, and E. Manning, "Modelling signal behavior in dame: A rule based designer of microprocessor based systems," in *Proceedings of the 1990 International Sympsium on Circuit and Systems*, (New Orleans, Louisiana), pp. 1497–1500, May 1990.

[24] H. Stone, *Microcomputer Interfacing*. Reading, Massachussetts: Addison-Wesley, 1982.

[25] M. A. Escalante, "Bus arbitration modelling and design in DAME: An expert microprocessor-based-systems designer," Master's thesis, University of Victoria, Department of Electrical and Computer Engineering, 1991.

[26] F. Rubin, "An improved algorithm for testing the planarity of a graph," *IEEE Transactions on Computers*, vol. c-24, Feb. 1975.

[27] G. Kant and H. Bodlaender, "Triangulating planar graphs while minimizing the maximum degree," in *Proc. 3rd Scandinavian Workshop on Algorithm Theory*, (Helsinki), pp. 258–271, July 1992.

[28] H. de Fraysseix, J. Pach, and R. Pollack, "How to draw a planar graph on a grid," *Combinatorica*, vol. 10, pp. 41–51, 1990.

[29] R. Tamassia, "Planar orthogonal drawings of graphs," in *Proc. IEEE Int. Symp. on Circuits and Systems*, 1990.

[30] L. Kucera, *Combinatorial Algorithms*. Adam Hilger, 1990.

[31] L. W. Beineke and R. J. Wilson, *Selected Topics in Graph Theory 3*. Academic Press Limited, 1988.

[32] W. T. Tutte, *Connectivity in Graphs*. University of Toronto Press, 1966.

Appendix A

DAME Shell Schemata

DAME user interface is built on top of Knowledge Craft's window, graphic and commonad systems. This appendix lists the frames associated with DAME shell.

A.1 DAME Shell

DAME shell is an icon manager, and also a shell icon command. It has one command, dame-shell-icon-window.

```
(defschema dame-shell :simple
  (has-commands dame-shell-icon-window)
  (initialize-actions srlc::icon-manager-initialize-action
                      srlc::shell-resume-action put-up-logo-fn)
  (resume-actions srlc::icon-manager-resume-action
                  srlc::shell-resume-action
                  dame-shell-resume-actions)
  (exit-actions srlc::icon-manager-exit-action
                standard-shell-exit-action)
  (icon-definition-type function)
  (icon-definition shell-icon-fcn)
  (width 50) (height 50)
  (display-name "Dame Shell")
  (extended-help "This is DAME Shell,
  a user interface for DAME"))
```

A.2 Dame-shell-icon-window Schema

Dame-shell-icon-window is an icon command window. It has six commands: p-edit, palm, listener, ops-workbench, toolbox, dame-editor. The first five commands are Knowledge Craft's build in work centers.

```
(defschema dame-shell-icon-window :simple
  (is-a icon-command-window)
  (has-commands p-edit palm listener ops-workbench
toolbox dame-editor)
  (ask-user-for-edges nil)
  (vertical-gap 10) (horixontal-gap 25)
  (xl 0) (yt 0) (xr 0.5) (yb 0.09))
```

A.3 DAME Editor Schema

Dame-editor is a task, and also a shell icon command. It has 12 global commands.

```
(defschema dame-editor :simple
  (is-a task shell-icon-command)
  (icon-definition-type function)
  (icon-definition dame-icon-fcn)
  (width 50) (height 50)
  (instance-prefix Dame)
  (display-name "Dame")
  (initialize-actions srlc::link-instance-to-generic-task
                      srlc::icon-task-initialize
                      dame-initialize)
  (exit-actions dame-exit-actions srlc::icon-task-exit
                srlc::unlink-instance-to-generic-task)
  (resume-actions srlc::icon-task-activate dame-resume-actions)
  (max-instances) (relation) (selected-font)
  (hierarchy) (path) (editing-schema)
  (selected-slot) (current-slot) (current-value)
  (value-position) (command-window) (command-viewport)
  (base-canvas) (base-window) (base-viewport)
  (io-canvas) (io-window) (io-viewport)
  (s-canvas) (s-window) (s-viewport)
  (sc-canvas) (sc-window) (sc-viewport)
  (h-canvas) (h-window) (h-viewport)
  (psc-canvas) (psc-window) (psc-viewport)
  (dsp-viewport)
  (display-action-graph-viewport)
  (component-name) (component-type)
  (package-name) (number-of-pin)
  (pin-item) (pin-items) (file-name)
  (delete-symbol) (update-command-default t)
  (has-commands)
  (task-global-commands dame-command-window
                        dame-mouse-scroll-up
                        dame-mouse-scroll-down
                        dame-mouse-r-1
                        select-slot-cmd
                        select-slot-content-cmd
                        slot-content-menu-cmd
                        select-permitted-slot-content-cmd
                        permitted-slot-content-menu-cmd
```

```
                        display-action-graph-menu-cmd
                        display-component-menu-cmd
                        component-pin-menu-cmd)
  (command-prompt "Dame Command")
  (extended-help ("This Dame Editor is used for creating and
   editing components.")))
```

A.4 Dame-editor's Global Commands

Dame-editor has 12 global commands. They are: dame-command-window, dame-mouse-scroll-up, dame-mouse-scroll-down, dame-mouse-r-1, select-slot-cmd, select-slot-content-cmd, slot-content-menu-cmd, select-permitted-slot-content-cmd, permitted-slot-content-menu-cmd, display-action-graph-menu-cmd, display-component-menu-cmd, component-pin-menu-cmd. Here lists all of them except the dame-command-window because this schema relates to all DAME editor commands. Dame-command-window is listed in appendix B.

```
(defschema dame-mouse-scroll-up :simple
  (instance simple-command)
  (inputs mouse-l-1-left-border)
  (actions dame-scroll-up-action))

(defschema dame-mouse-scroll-down :simple
  (instance simple-command)
  (inputs mouse-r-1-left-border)
  (actions dame-scroll-down-action))

(defschema dame-mouse-r-1 :simple
(instance simple-command)
(inputs mouse-r-1)
(predicate dame-mouse-r-1-predicate)
(actions dame-mouse-r-1-actions))

(defschema select-slot-cmd :simple
  (instance simple-command)
  (inputs mouse-l-1)
  (predicate select-slot-predicate)
  (short-help "Select Slot Help")
  (mouse-line-help " Position mouse on an item and enter
     Mouse-R-1 to select it")
  (actions select-slot-cmd-actions))

(defschema select-slot-content-cmd :simple
  (instance simple-command)
  (inputs mouse-l-1)
  (predicate select-slot-content-predicate)
  (actions select-slot-content-cmd-actions))

(defschema slot-content-menu-cmd :simple
  (instance simple-command)
  (inputs mouse-r-1)
```

```
  (predicate select-slot-content-predicate)
  (actions slot-content-menu-cmd-actions))

(defschema edit-slot-value-cmd :simple
  (instance complex-command)
  (inputs "Edit slot value")
  (has-commands get-slot-content-name-cmd)
  (entry-actions edit-slot-value-entry)
  (exit-actions change-slot-content-name-actions))

(defschema get-slot-content-name-cmd :simple
  (instance simple-command)
  (inputs slot-value-input)
  (actions get-slot-value-name-action))

(defschema select-permitted-slot-content-cmd :simple
  (instance simple-command)
  (inputs mouse-l-1)
  (predicate select-permitted-slot-content-predicate)
  (actions select-permitted-slot-content-cmd-actions))

(defschema permitted-slot-content-menu-cmd :simple
  (instance simple-command)
  (inputs mouse-r-1)
  (predicate select-permitted-slot-content-predicate)
  (actions permitted-slot-content-menu-cmd-actions))

(defschema copy-permitted-slot-content-cmd :simple
  (instance complex-command)
  (inputs "copy slot content name")
  (has-commands get-slot-content-name-cmd)
  (entry-actions copy-permitted-slot-content-entry)
  (exit-actions copy-permitted-slot-content-actions))

(defschema display-action-graph-menu-cmd :simple
  (instance simple-command)
  (inputs mouse-r-1)
  (predicate display-action-graph-menu-cmd-predicate)
  (actions display-action-graph-menu-cmd-actions))

(defschema display-component-menu-cmd :simple
  (instance simple-command)
  (inputs mouse-r-1)
  (predicate graphic-component-viewport-predicate)
  (actions display-component-menu-cmd-actions))

(defschema component-pin-menu-cmd :simple
(instance simple-command)
(inputs mouse-l-1)
(predicate graphic-component-viewport-predicate)
(actions grph-menu-actions))

(defschema component-pin-menu-cmd-group :simple
(instance command-group)
(has-commands signal-coconut-cmd delete-signal-cmd))
```

```
(defschema signal-coconut-cmd :simple
(instance simple-command)
(inputs "edit signal schema (coconut)")
(actions signal-coconut-actions))

(defschema signal-name-cmd :simple
  (instance complex-command)
  (inputs "Signal Name")
  (command-prompt "Signal Name")
  (has-commands get-signal-name))

(defschema get-signal-name :simple
  (instance simple-command)
  (inputs signal-name-input)
  (actions get-signal-name-actions))

(defschema signal-name-input :simple
  (instance read-input)
  (convert convert-to-symbol)
  (read-prompt "Signal Name")
  (update-read-default nil))

(defschema delete-signal-cmd :simple
(instance complex-command)
(inputs "delete signal schema")
(exit-actions delete-signal-exit-actions)
(has-commands delete-confirm-cmd))

(defschema delete-confirm-cmd :simple
(instance simple-command)
(inputs delete-confirm-input)
(actions get-delete-confirm-actions))

(defschema delete-confirm-input :simple
(instance read-input)
(convert convert-to-symbol)
(read-prompt "Do you really want to delete?")
(update-read-defaule nil))
```

Appendix B

DAME Commands Schemata

This appendix lists schemata for DAME commands.

B.1 Dame-command-window and dame-command Schemata

Dame-command-window is a icon-command-window. It has the following icon commands: save-icon, load-icon, create-component-icon, edit-schema-icon, add-slot-cion, add-value-icon, return-icon, display-component-icon, exit-icon. Each of them corresponds to a command schema.

```
(defschema dame-command-window :simple
  (simple-window t)
  (ask-user-for-edges nil)
  (xl 0.9) (yt 0.1) (xr 1.0) (yb 1.0)
  (width 74) (height 31)
  (wraparound t)
  (viewport-name "")
  (find-item-p t)
  (create-command-instance-p t)
  (mouse-line-help "Click left to select an icon")
  (is-a icon-command-window)
  (has-commands save-icon load-icon
                create-component-icon
                edit-schema-icon
                add-slot-icon
                add-value-icon
                return-icon
                display-component-icon
                exit-icon))

(defschema dame-command-group :simple
  (instance command-group)
  (sub-command-of dame)
```

```
  (has-commands return-cmd save-cmd load-cmd
                create-component-cmd
                display-component-cmd
                edit-schema-cmd add-slot-cmd
                add-value-cmd exit-cmd))
```

B.2 Save Command Schemata

```
(defschema return-cmd :simple
  (instance simple-command)
  (inputs "RETURN")
  (actions))

(defschema save-icon :simple
  (is-a icon-command simple-command)
  (icon-definition-type generic-icon)
  (icon-definition "SAVE")
  (width 70)
  (height 30)
  (inputs MOUSE-L-1)
  (actions save-icon-actions)
  (icon-help-title "Help for Save Command")
  (extended-help ("Save Command is used to save schemata"
  "Input filename")))

(defschema save-cmd :simple
  (instance complex-command)
  (inputs "SAVE")
  (has-commands get-save-file-name-cmd)
  (entry-actions save-cmd-entry)
  (post-sub-command-actions save-actions)
  (exit-actions invert-icon-to-white))

(defschema get-save-file-name-cmd :simple
  (instance simple-command)
  (inputs file-name-input)
  (actions get-save-file-name-action))

(defschema file-name-input :simple
  (instance read-input)
  (convert convert-to-symbol)
  (read-prompt "File Name"))

(defschema dame-save-manager
  (instance save-manager))
```

B.3 Load Command Schemata

```
(defschema load-icon :simple
```

```
  (is-a icon-command simple-command)
  (icon-definition-type generic-icon)
  (icon-definition "LOAD")
  (width 70) (height 30)
  (inputs MOUSE-L-1)
  (actions load-icon-actions)
  (icon-help-title "Help for Load Command")
  (extended-help ("Load command is used to load schemada."
  "Input filename.")))

(defschema load-cmd :simple
  (instance complex-command)
  (inputs "LOAD")
  (has-commands get-load-file-name-cmd)
  (post-sub-command-actions load-file-action)
  (exit-actions invert-icon-to-white))

(defschema get-load-file-name-cmd :simple
  (instance simple-command)
  (inputs file-name-input)
  (actions get-load-file-name-action))
```

B.4 Create Component Command Schemata

```
(defschema create-component-icon :simple
  (is-a icon-command simple-command)
  (icon-definition-type generic-icon)
  (icon-definition "CREATE" "COMPONENT")
  (width 70) (height 30)
  (inputs mouse-l-1)
  (actions create-component-icon-actions)
  (icon-help "Help for Create Component Command")
  (extended-help ("Create Component Command is used to"
  "create a new component")))

(defschema create-component-cmd :simple
  (instance complex-command)
  (inputs "create-component")
  (has-commands get-new-component-name-cmd)
  (post-sub-command-actions create-component)
  (exit-actions invert-icon-to-white))

(defschema get-new-component-name-cmd :simple
  (instance complex-command)
  (inputs schema-name-input)
  (entry-actions get-component-name-entry)
  (has-commands get-number-of-pin-cmd))

(defschema schema-name-input :simple
  (instance read-input)
  (convert convert-to-symbol)
  (read-prompt "Schema Name")
```

```
  (update-read-default t))

(defschema get-number-of-pin-cmd :simple
  (instance simple-command)
  (inputs pin-number-input)
  (actions get-number-of-pin-actions))

(defschema pin-number-input :simple
  (instance read-input)
  (convert convert-to-number)
  (read-prompt "Number of pins")
  (error-actions prompt-error-message))
```

B.5 Edit-schema Command Schemata

```
(defschema edit-schema-icon :simple
  (is-a icon-command simple-command)
  (icon-definition-type generic-icon)
  (icon-definition "EDIT" "COMPONENT")
  (width 70) (height 30)
  (inputs MOUSE-L-1)
  (actions edit-schema-icon-actions)
  (icon-help-title "Help for Create Component Command")
  (extended-help ("SCHEMA command is used to create a new
   schema, or" "to edit a schema from the
                root")))

(defschema edit-schema-cmd :simple
  (instance complex-command)
  (entry-actions edit-schema-cmd-entry)
  (inputs "EDIT SCHEMA")
  (has-commands get-schema-name-cmd)
  (post-sub-command-actions replace-base-viewport)
  (exit-actions invert-icon-to-white))

(defschema get-schema-name-cmd :simple
  (instance complex-command)
  (entry-actions get-schema-name-action check-schema-action)
  (inputs schema-name-input)
  (has-commands create-schema-cmd))

(defschema create-schema-cmd :simple
  (instance simple-command)
  (inputs create-schema-input)
  (actions create-schema-cmd-actions))

(defschema create-schema-input :simple
  (instance read-input)
  (convert convert-to-symbol)
  (read-prompt "The schema doesn't exist. Create?(y/n)")
  (update-read-default t))
```

B.6 Add-slot Command Schemata

```
(defschema add-slot-icon :simple
  (is-a icon-command simple-command)
  (icon-definition-type generic-icon)
  (icon-definition "ADD SLOT")
  (width 70) (height 30)
  (inputs MOUSE-L-1)
  (actions add-slot-icon-actions)
  (icon-help-title "help for Add-Slot command")
  (extended-help ("When a new slot has to be added into the
   schema," "click MOUSE-L-1.")))

(defschema add-slot-cmd :simple
  (instance complex-command)
  (entry-actions add-slot-cmd-entry)
  (inputs "ADD SLOT")
  (has-commands get-slot-name-cmd)
  (exit-actions invert-icon-to-white))

(defschema get-slot-name-cmd :simple
  (instance simple-command)
  (inputs slot-name-input)
  (actions get-slot-name-cmd-actions))

(defschema slot-name-input :simple
  (instance read-input)
  (convert convert-to-symbol)
  (read-prompt "Slot Name")
  (update-read-default t))
```

B.7 Add-Value Command Schemata

```
(defschema add-value-icon :simple
  (is-a icon-command simple-command)
  (icon-definition-type generic-icon)
  (icon-definition "ADD VALUE")
  (width 70) (height 30) (inputs mouse-l-1)
  (actions add-value-icon-actions)
  (icon-help-title "help for Add-Value command")
  (extended-help ("When a new value has to be added into
   the slot," "clicl MOUSE-L-1.")))

(defschema add-value-cmd :simple
  (instance complex-command)
  (entry-actions add-value-cmd-entry)
  (inputs "ADD VALUE")
  (has-commands get-value-name-cmd)
  (post-sub-command-actions add-value-post)
  (exit-actions invert-icon-to-white))
```

```
(defschema get-value-name-cmd :simple
  (instance simple-command)
  (inputs slot-value-input)
  (actions get-value-name-action))

(defschema slot-value-input :simple
  (instance read-input)
  (read-prompt "Content name"))

(defschema exit-cmd :simple
  (instance simple-command)
  (inputs "EXIT") (actions))
```

B.8 Return Command Schemata

```
(defschema return-icon :simple
  (is-a icon-command simple-command)
  (icon-definition-type generic-icon)
  (icon-definition "RETURN")
  (width 70) (height 30) (inputs MOUSE-L-1)
  (actions return-icon-actions)
  (icon-help-title "Return to previous edit.")
  (extended-help ("Return Command is used to return the edit"
                  "to the previous edit window")))
```

B.9 Display-component Command Schemata

```
(defschema display-component-icon :simple
  (is-a icon-command simple-command)
  (icon-definition-type generic-icon)
  (icon-definition "DISPLAY" "COMPONENT")
  (width 70) (height 30) (inputs mouse-l-1)
  (actions display-component-icon-actions)
  (icon-help-title "Help for Display Component Command")
  (extended-help ("When selecting the DISPLAY COMPONENT command,
   the user" "placed in a mode for displaying
                  component")))

(defschema display-component-cmd :simple
  (instance simple-command)
  (inputs "Display Component")
  (actions display-component-actions))
```

B.10 Exit Command Schemata

```
(defschema exit-icon :simple
```

```
(is-a icon-command simple-command)
(icon-definition-type generic-icon)
(icon-definition "EXIT")
(width 70) (height 30) (inputs MOUSE-L-1)
(actions exit-icon-actions)
(icon-help-title "Help for Exit Command")
(extended-help ("When you want to exit dame,"
"click MOUSE-L-1.")))
```

Appendix C

Functions Related to DAME Shell Schemata

This appendix lists the lisp code functions related with DAME shell schemata listed in appendix A.

C.1 Functions in dame-shell schema

```
;;; Name: dame-shell-resume-actions
;;; Purpose: bury terminal window
(defun dame-shell-resume-actions (schema slot context)
  (bury-terminal-window))

;;;Name: shell-icon-fcn
;;;Purpose: This function creates dame-shell icon.
(defun shell-icon-fcn (schema canvas x y width height)
;;;Create the shcema of the circle in dame-shell icon.
        (cschema 'shell-circle :simple
                ('item-type 'circle)
                ('x 50)('y 35)('radius 20))
        (create-item 'shell-circle canvas)
;;;Create the schema of the box of dame-shell icon.
        (cschema 'shell-box :simple
                ('item-type 'box)
                ('xl 25)('yt 10)('xr 75)('yb 60))
        (create-item 'shell-box canvas)
        (cschema 'shell-triangle1 :simple
                ('item-type 'triangle)
                ('x1 3) ('y1 12) ('x2 12) ('y2 3) ('x3 12) ('y3 12))
        (create-item 'shell-triangle1 canvas)
;;;Create the schema of the text "shell" in dame-shell icon.
        (cschema 'shell-text1 :simple
                ('item-type 'text-string)
                ('x 34) ('y 33)
```

```
                ('font 'medium-roman)
                ('text "DAME"))
        (create-item 'shell-text1 canvas)
        (cschema 'shell-text2 :simple
                ('item-type 'text-string)
                ('x 33) ('y 45)
                ('font 'screen-12)
                ('text "shell"))
        (create-item 'shell-text2 canvas)
;;;Return the items list as the values of this function.
        (list 'shell-circle 'shell-box 'shell-text1 'shell-text2
                'shell-triangle1)
) ;;;End of function shell-icon-fcn.
```

C.2 Functions in dame-editor schema

```
;;;Name: dame-icon-fcn
;;;Purpose: This function creates the dame task icon.
(defun dame-icon-fcn (schema canvas x y width height)
;;;Create the schema of the box of dame icon.
        (cschema 'dame-box :simple
                ('item-type 'box)
                ('xl x)('yt y)('xr (+ x width))('yb (+ y height)))
        (create-item 'dame-box canvas)
;;;Create the schema of the ellipse of dame icon.
        (cschema 'dame-ellipse :simple
                ('item-type 'ellipse)
                ('x (+ x 25))('y (+ y 28))
                ('x-radius 20)
                ('y-radius 13))
        (create-item 'dame-ellipse canvas)
;;;Create the schema of the polygon of dame icon.
        (cschema 'dame-polygon :simple
                ('item-type 'polyline)
                ('vertices (cons (+ x 25)(+ y 2))
                                (cons (+ x 1)(+ y 25))
                                (cons (+ x 13)(+ y 49))
                                (cons (+ x 37)(+ y 49))
                                (cons (+ x 49)(+ y 25))
                                (cons (+ x 25)(+ y 2))))
        (create-item 'dame-polygon canvas)
        (cschema 'dame-triangle :simple
                ('item-type 'triangle)
                ('x1 (+ x 12)) ('y1 (+ y 6))
                ('x2 (+ x 38)) ('y2 (+ y 6))
                        ('x3 (+ x 25)) ('y3 (+ y 13)))
              (create-item 'dame-triangle canvas)
       (cschema 'dame-small-ellipse :simple
                        ('item-type 'ellipse)
              ('x (+ x 25)) ('y (+ y 45))
              ('x-radius 10) ('y-radius 3))
       (create-item 'dame-small-ellipse canvas)
```

```
;;;Create the schema of the text "dame" in dame-shell icon.
        (cschema 'dame-text :simple
                ('item-type 'text)
                ('x (+ x 8))
                ('y (+ y 32))
                ('font 'bold-roman)
                ('text "DAME"))
        (create-item 'dame-text canvas)
;;;Return the items list as the values of this function.
        (list 'dame-box 'dame-ellipse 'dame-text 'dame-polygon
                'dame-triangle 'dame-small-ellipse)
) ;;;End of function dame-icon-fcn.

;;;;;;;;;;;;;;;; Begin dame-initialize ;;;;;;;;;;;;;;;;;;;

;;; Name: dame-initialize
;;; Purpose: to create dame editor windows.
(defun dame-initialize (task &rest x)
;;; create a link to parent task.
  (hierarchy-from-parent)
;;; create the four viewports for displaying slots, slot contents,
;;; hierarchy, and permitted slot contents.
  (create-viewports)
;;; create command window at the right side of the editor window.
  (create-cmd-window)
;;; create the command input window.
  (create-io-viewport task))

;;; Name: create-viewport
;;; Purpose: to create DAME editor's displying viewports.
(defun create-viewports ()
  (create-base-viewport)
  (create-s-viewport)   ; create slot viewport
  (create-sc-viewport)  ; create slot content viewport
  (create-h-viewport)   ; create hierarchy viewport
  (create-psc-viewport) ; create permitted slot contents viewport
)

;;; Name: hierarchy-from-parent
;;; Purpose: create a link to the parent task.
(defun hierarchy-from-parent ()
  (let ((parent (get-value *current-task* 'parent-task)))
    (cond ((equal (get-value parent 'instance) 'dame)
           (new-value *current-task* 'editing-schema
             (get-selected-text (get-value parent 'sc-viewport)))
           (new-values *current-task* 'hierarchy
             (append (get-values parent 'hierarchy)
               (list (get-value parent 'editing-schema))))
           (new-values *current-task* 'path
             (append (get-values parent 'path)
                     (list (get-value parent 'selected-slot))
             ))))))

;;; Name: get-selected-text
;;; Purpose: to get the text string from the viewport.
```

```
;;; Input: the viewport name.
(defun get-selected-text (viewport)
  (get-value (get-value viewport 'current-item) 'text-name))

;;;Name: create-canvas-window-viewport-fn
;;;Purpose: to create canvas, window and viewport.
;;;Input: viewport-name,
;;;         slots to store the names of canvas, window and viewport,
;;;         i.e. canvas-slot, window-slot, viewport-slot.
(defun create-canvas-window-viewport-fn
       (viewport-name canvas-slot window-slot
        viewport-slot xl yt xr yb)
  (let* ((canvas (create-canvas (symgen 'canvas)))
         (window (create-window (symgen 'window) canvas 0 0
                                (* (- xr xl) 100) (* (- yb yt) 100) ))
         (viewport (create-viewport (cschema (symgen 'viewport)
     :simple
                                    ('ask-user-for-edges nil)
                                    ('viewport-name viewport-name)
                                    ('xl xl)('yt yt)('xr xr)('yb yb)
                                    ('has-left-scroll-bar t)
                                    ('current-item))
                         window)))
        (turn-on-scroll-bars viewport)
        (new-value *current-task* canvas-slot canvas)
        (new-value *current-task* window-slot window)
         (new-value *current-task* viewport-slot viewport) ))

;;; Name: create-canvas-window-viewport-without-bar
;;;Purpose: to create canvas, window and viewport without scroll
;;;    bars.
;;;Input: viewport-name,
;;;         slots to store the names of canvas, window and viewport,
;;;         i.e. canvas-slot, window-slot, viewport-slot.
(defun create-canvas-window-viewport-without-bar
       (viewport-name canvas-slot window-slot
        viewport-slot xl yt xr yb)
  (let* ((canvas (create-canvas (symgen 'canvas)))
         (window (create-window (symgen 'window) canvas 0 0
                                (* (- xr xl) 100) (* (- yb yt) 100) ))
         (viewport (create-viewport (cschema (symgen 'viewport)
      :simple
                                    ('ask-user-for-edges nil)
                                    ('viewport-name viewport-name)
                                    ('xl xl)('yt yt)('xr xr)('yb yb))
                         window)))
        (new-value *current-task* canvas-slot canvas)
        (new-value *current-task* window-slot window)
         (new-value *current-task* viewport-slot viewport) ))

;;; Name: create-base-viewport
;;; Purpose: to create a base-viewport.
(defun create-base-viewport ()
  (let ((editing-schema (get-value *current-task* 'editing-schema))
        (title nil))
```

```
    (if editing-schema
      (setq title (get-title editing-schema))
      (setq title "WHICH SCHEMA DO YOU WANT TO EDIT?"))
    (create-canvas-window-viewport-without-bar
      title 'base-canvas 'base-window 'base-viewport 0 0.1 1 1)))

;;; Name: get-title
;;; Purpose: to get the current editing schema name.
(defun get-title (editing-schema)
  (let ((is-a nil)
        (copy-of nil))
    (cond ((slotp editing-schema 'is-a)
           (setq is-a (get-value editing-schema 'is-a)))
          ((slotp editing-schema 'copy-of)
           (setq copy-of (get-value editing-schema 'copy-of))))
    (format nil "EDITING SCHEMA: ã    IS A: ã    COPY-OF: ã"
      editing-schema is-a copy-of)))

;;; Name: create-s-viewport
;;; Purpose: to create a viewport for displaying slots
(defun create-s-viewport ()
  (create-canvas-window-viewport-fn "SLOTS"
    's-canvas 's-window 's-viewport 0 0.14 0.225 0.5)
  (let ((schema (get-value *current-task* 'editing-schema)))
    (cond ((schemap schema)
           (slot-viewport-redisplay)))))

;;; Name: create-sc-viewport
;;; Purpose: to create a viewport for displaying slot contents.
(defun create-sc-viewport ()
  (create-canvas-window-viewport-fn "SLOT-CONTENT"
    'sc-canvas 'sc-window 'sc-viewport 0.225 0.14 0.9 0.5))

;;; Name: create-h-viewport
;;; Purpose: to create a viewport for displaying hierarchy.
(defun create-h-viewport ()
  (create-canvas-window-viewport-fn "HIERARCHY"
    'h-canvas 'h-window 'h-viewport 0 0.5 0.225 0.9)
  (let ((hierarchy (get-values *current-task* 'hierarchy))
        (editing-schema (get-value *current-task* 'editing-schema)))
    (cond ((car hierarchy)
           (create-items (append hierarchy (list editing-schema))
                         (get-value *current-task* 'h-canvas)
             (get-value *current-task* 'h-viewport) 1 5)))))

;;; Name: create-psc-viewport
;;; Purpose: to create a viewport for displaying permitted slot
;;;          contents.
(defun create-psc-viewport ()
  (create-canvas-window-viewport-fn "PERMITTEN-SLOT-CONTENTS"
    'psc-canvas 'psc-window 'psc-viewport 0.225 0.5 0.9 0.9))

;;; Name: create-cmd-window
;;; Purpose: to create command window at the right side of the
;;;          editor window.
```

```
(defun create-cmd-window ()
  (let* ((cmd-window (create-schema
          (symgen 'dame-command-window) :simple t)))
    (add-value *current-task* 'task-global-commands cmd-window
      :inherit-values t)
    (create-slot cmd-window 'instance)
    (new-value cmd-window 'instance 'dame-command-window)
    (new-values cmd-window 'has-commands
      (get-values 'dame-command-window 'has-commands))
    (call-method cmd-window 'create-window)
    (new-value *current-task* 'command-window cmd-window)
    (new-value *current-task* 'command-viewport
      (get-value cmd-window 'icon-viewport))))

;;;Name: create-io-viewport
;;;Purpose: to create DAME I/O command prompt viewport.
(defun create-io-viewport (task)
  (multiple-value-bind (canvas window viewport)
     (create-canvas-window-viewport :xleft 0 :ytop 0.9
        :xright 0.9 :ybottom 1)
;;;Store the names of canvas, window, and viewport of prompt
;;;viewport in the current instance schema of dame.
    (new-value task 'io-canvas canvas)
    (new-value task 'io-window window)
    (new-value task 'io-viewport viewport)
;;;Create an input-interface schema for the current task and store
;;;the name of the interactive prompt viewport in it.
    (let ((input-interface (gentemp)))
      (cschema input-interface :simple
              ('instance 'input-interface))
      (new-value task 'input-interface input-interface)
      (new-value input-interface 'textio-viewport viewport))))

;;;;;;;;;;;;;; End dame-initialize;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;; Begin dame-exit-actions;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;Name: dame-exit-actions
;;;Purpose: This function destroys the canvases
;;;           created by one instance of dame.
(defun dame-exit-actions (schema slot context)
  (let ((child-tasks (get-values schema 'child-tasks)))
    (cond ((not (equal nil child-tasks))
          (dolist (task child-tasks)
            (call-method task 'kill)))))
;;;Destroy the canvases.
  (destroy-canvas (get-value *current-task* 'io-canvas))
  (destroy-canvas (get-value *current-task* 'psc-canvas))
  (destroy-canvas (get-value *current-task* 'h-canvas))
  (destroy-canvas (get-value *current-task* 'sc-canvas))
  (destroy-canvas (get-value *current-task* 's-canvas))
  (if (not (equal (get-value *current-task* 'dsp-viewport) nil))
    (destroy-canvas (get-value
                     (get-value
                      (get-value *current-task* 'dsp-viewport)
```

```
                    'on-window)
                   'on-canvas)))
;;;Destroy icon command windows.
  (call-method
    (get-value *current-task* 'command-window) 'destroy-window)
  (destroy-canvas (get-value *current-task* 'base-canvas)))

;;;;;;;;;;;;;;;;;; End dame-exit-actions;;;;;;;;;;;;;;;;;;;;

;;; Name: dame-resume-actions
;;; Purpose: to expose editor windows when it is resumed.
(defun dame-resume-actions (schema slot context)
  (expose-edit schema))

;;; Name: expose-edit
;;; Purpose: to expose viewports of dame editor.
(defun expose-edit (task)
  (expose-viewport (get-value task 'base-viewport))
  (expose-viewport (get-value task 'command-viewport))
  (expose-viewport (get-value task 's-viewport))
  (expose-viewport (get-value task 'sc-viewport))
  (expose-viewport (get-value task 'h-viewport))
  (expose-viewport (get-value task 'psc-viewport))
  (expose-viewport (get-value task 'io-viewport))
  (let ((dsp-viewport (get-value task 'dsp-viewport)))
    (cond ((not (equal dsp-viewport nil))
           (expose-viewport dsp-viewport)
          (delete-any-values dsp-viewport 'buried))))
  (let ((graph-viewports (get-values task
    'display-action-graph-viewport)))
    (cond (graph-viewports
           (pop-up-message*
            (format nil "this is in expose-edit ~S" graph-viewports))
          (dolist (vpt graph-viewports)
            (if (schemap vpt) (expose-viewport vpt)))))))
```

C.3 Functions Related to DAME editor's Global Commands

```
;;;Name: dame-scroll-up-action
;;;Purpose: This function is the actions of dame-mouse-scroll-up
;;;         command. Left click on scroll bar scrolls up.
(defun dame-scroll-up-action (schema slot context)
  (let ((window (get-value (viewport-under-mouse) 'on-window)))
    (scroll-window window 0 10)))

;;;Name: dame-scroll-down-action
;;;Purpose: This function is the actions of dame-mouse-scroll-down
;;;         command. Right click on scroll bar scrolls text down.
(defun dame-scroll-down-action (schema slot context)
  (let ((window (get-value (viewport-under-mouse) 'on-window)))
    (scroll-window window 0 -10)))
```

```
;;; Name: dame-mouse-r-1-predicate
;;; Purpose: to see if the mouse click is in slot viewport
(defun dame-mouse-r-1-predicate (schema slot context)
  (cond ((equal (viewport-under-mouse)
                (get-value *current-task* 's-viewport))
         t)))

;;;;;;;;;;;;; Begin dame-mouse-r-1-actions ;;;;;;;;;;;;;;;

;;; Name: dame-mouse-r-1-actions
;;; Purpose: if the mouse righ butom is clicked on an graphic item,
;;;      display the pop-up menu that is related to this item
(defun dame-mouse-r-1-actions (schema slot context)
  (setq viewport (viewport-under-mouse) x-y-l (multiple-value-list
 (get-current-mouse-position)))
  (calculate-x-y-fn viewport (car x-y-l) (cadr x-y-l))
  (cond ((find-item viewport xc yc)
         (call-method 'pop-up-local-menu 'actions))
        (t (beep-viewport (get-value *current-task* 'io-viewport))))
  (call-method 'task-manager 'exit-into *current-task*))

;;;Name: calculate-x-y-fn
;;;Purpose: This function calculates the position coordinates on
;;;         the canvas acording to the coordinates on the screen.
(defun calculate-x-y-fn (viewport x y)
       (let*  ((vp-xl (get-value viewport 'inside-xl))
               (vp-xr (get-value viewport 'xr))
               (vp-yt (get-value viewport 'inside-yt))
               (vp-yb (get-value viewport 'yb))
               (window (get-value viewport 'on-window))
               (wd-xl (get-value window 'xl))
               (wd-xr (get-value window 'xr))
               (wd-yt (get-value window 'yt))
               (wd-yb (get-value window 'yb)))
         (setq xc (* (- x vp-xl) (/ (- wd-xr wd-xl) (- vp-xr vp-xl)))
           yc (* (- y vp-yt) (/ (- wd-yb wd-yt) (- vp-yb vp-yt))))))

;;;;;;;;;;;;;;; Eng dame-mouse-r-1-actions ;;;;;;;;;;;;;;;;;;;

;;; Name: select-slot-predicate
;;; Purpose: check if the mouse click is in slot viewport
(defun select-slot-predicate (schema slot context)
  (cond ((equal (viewport-under-mouse)
                (get-value *current-task* 's-viewport))
         t)
        (t nil)))

;;;;;;;;;;;;; Begin select-slot-cmd-actions ;;;;;;;;;;;;;;;;;;

;;; Name: select-slot-cmd-actions
;;; Purpose: to inverte the background color of the selected slot,
;;;          to display the selected slot contents and its
;;;          permitted contents.
(defun select-slot-cmd-actions (schema slot context)
  (let* ((input-event (call-method 'task-manager 'last-input-event))
```

```
        (viewport (get-value *current-task* 's-viewport))
        (new-item (find-item viewport (input-event-x input-event)
                                      (input-event-y input-event)))
        (item (get-value viewport 'current-item)))
    (if new-item
      (let ((slot-name (get-value new-item 'text-name)))
        (cond ((equal new-item item)
               (display-slot-contents slot-name)
               (display-permitted-slot-contents slot-name))
              (t (if item (invert-item-background item viewport))
                 (invert-item-background new-item viewport)
                 (new-value viewport 'current-item new-item)
                 (new-value *current-task* 'selected-slot slot-name)
                 (display-slot-contents slot-name)
                 (display-permitted-slot-contents slot-name))))
        (beep-viewport viewport)))
  (call-method 'task-manager 'exit-into *current-task*))

;;; Name: display-slot-contents
;;; Purpose: to display slot contents
;;; Input: slot name
(defun display-slot-contents (slot-name)
  (let ((canvas (get-value *current-task* 'sc-canvas))
        (viewport (get-value *current-task* 'sc-viewport))
        (schema (get-value *current-task* 'editing-schema)))
    (delete-items canvas viewport)
    (create-items (get-values schema slot-name :path nil)
canvas viewport 1 5)))

;;; Name: create-items
;;; Purpose: to create graphic items.
;;; Input: list is a list of text strings, canvas is the name
;;;        of the canvas the graphic items created on, viewport
;;;        is the viewport name the graphic items displayed on,
;;;        x, y are the cordinates the graphic items placed.
(defun create-items (list canvas viewport x y)
  (dolist (element list y)
    (display-item (create-item (cschema (unique-name) :simple
                               ('item-type 'text)
                               ('x x)('y y)
                               ('text-name element)
                               ('text (format nil "~S" element)))
                  canvas) viewport)
    (setq y (+ y 3))))

;;; Name: delete-items
;;; Purpose: to delete all the graphic items on the canvas.
;;; Input: canvas is the canvas name,
;;;           viewport is the viewport name.
(defun delete-items (canvas viewport)
  (let ((items (get-values canvas 'canvas-items))
        (black-item (get-value viewport 'current-item)))
    (cond ((car items)
           (dolist (element items)
             (if (equal element black-item)
```

```
              (invert-item-background element viewport))
             (erase-item element viewport)
             (destroy-item element))))))

;;; Name: display-permitted-slot-contents
;;; Purpose: to display the permitted contents of the selected slot.
(defun display-permitted-slot-contents (slot-name)
  (let ((canvas (get-value *current-task* 'psc-canvas))
        (viewport (get-value *current-task* 'psc-viewport))
        (schema (get-value *current-task* 'editing-schema))
        (root (get-value *current-task* 'hierarchy))
        (path (append (get-values *current-task* 'path)
                      (list *current-task* 'selected-slot)))
        (y 5)
        (x 1)
        (contents nil))
    (delete-items canvas viewport)
    (dolist (element (get-permitted-slot-contents schema slot-name))
      (setq y (create-items (list element) canvas viewport 1 y))
      (setq contents (get-contents (list element) 'is-a+inv))
      (when contents
       (create-items (list 'is-a+inv) canvas viewport 1 y)
        (setq y (create-items contents canvas viewport 10 y)))
      (setq contents (get-contents (push element contents)
       'copy-of+inv))
      (when contents
       (create-items (list 'copy-of+inv) canvas viewport 1 y)
        (setq y (create-items contents canvas viewport 10 y))))))

;;; Name: get-permitted-slot-contents
;;; Purpose: to get the permitted contents from the corresponding
;;;          template.
;;; Input: the editing schema name and the selected slot name.
(defun get-permitted-slot-contents (schema slot)
  (if (and (local-slot-p schema 'copy-of)
           (schemap (get-value schema 'copy-of))
           (local-slot-p (get-value schema 'copy-of) slot))
    (get-values (get-value schema 'copy-of) slot :path nil)
    nil))

;;; Name: get-contents
;;; Purpose: to get the contents of a slot from a list of schemata.
(defun get-contents (schema-list slot)
  (let ((result nil))
    (dolist (element schema-list result)
      (if (and (schemap element) (local-slot-p element slot))
       (setq result (append (get-values element slot) result))))))

;;;;;;;;;;;;;; End select-slot-cmd-actions ;;;;;;;;;;;;;;;;;;

;;; Name: select-slot-content-predicate
;;; Purpose: to check if the mouse is in the slot content viewport
(defun select-slot-content-predicate (schema slot context)
  (cond ((equal (viewport-under-mouse)
                (get-value *current-task* 'sc-viewport))
```

```
        t)
       (t nil)))

;;; Name: selecte-slot-content-cmd-actions
;;; Purpose: to select the content itme , and invert the background
;;;          color of the selecte item.
(defun select-slot-content-cmd-actions (schema slot context)
  (select-item))

;;; Name: select-item
;;; Purpose: to select an item.
(defun select-item ()
  (let* ((input-event (call-method 'task-manager 'last-input-event))
         (viewport (viewport-under-mouse))
         (new-item (find-item viewport (input-event-x input-event)
                                       (input-event-y input-event)))
          (item (get-value viewport 'current-item)))
    (if new-item
      (cond ((equal new-item item) nil)
            (t (if (schemap item)
                   (invert-item-background item viewport))
               (invert-item-background new-item viewport)
               (new-value viewport 'current-item new-item)))
      (beep-viewport viewport))))

;;; Name: slot-content-menu-cmd-actions
;;; Purpose: to select a command from a pop-up menu.
(defun slot-content-menu-cmd-actions (schema slot context)
  (select-item-menu 'slot-content-menu)
  (call-method 'task-manager 'exit-into *current-task*))

;;; Name: select-item-menu
;;; Purpose: to select an item and display its related pop-up menu.
(defun select-item-menu (menu-name)
  (let* ((input-event (call-method 'task-manager 'last-input-event))
         (viewport (viewport-under-mouse))
         (item (get-value viewport 'current-item))
         (new-item (find-item viewport (input-event-x input-event)
                                       (input-event-y input-event))))
    (when new-item
      (cond ((equal item new-item) nil)
            (t (if (schemap item)
                   (invert-item-background item viewport))
               (invert-item-background new-item viewport)
               (new-value viewport 'current-item new-item)))
      (pop-up-menu menu-name))))

;;;;;;;;;;;;;;; Begin slot-content-menu ;;;;;;;;;;;;;;;;;;;;;;;;

;;; Name: make-slot-content-menu
;;; Purpose: to make a pop-up menu for slot content viewport.
(defun make-slot-content-menu ()
  (make-new-menu 'slot-content-menu 'momentary)
  (add-menu-slot 'slot-content-menu "Slot Content Menu"
    :not-selectable t)
```

```
  (add-menu-slot 'slot-content-menu " " :not-selectable t)
  (add-menu-slot 'slot-content-menu "EDIT VALUE"
    :item-action 'function :item-form 'edit-slot-value)
  (add-menu-slot 'slot-content-menu "EDIT SCHEMA"
    :item-action 'function :item-form 'edit-slot-schema)
  (add-menu-slot 'slot-content-menu "DELETE VALUE"
    :item-action 'function :item-form 'delete-slot-content)
  (menu-complete 'slot-content-menu))

(make-slot-content-menu)

;;; Name: edit-slot-value
;;; Purpose: to call edit-slot-value command.
(defun edit-slot-value ()
  (call-method 'task-manager 'queue-command 'edit-slot-value-cmd))

;;; Name: edit-slot-value-entry
;;; Purpose: to get the selected content string.
(defun edit-slot-value-entry (schema slot context)
  (call-method *current-task* 'set-current-input-buffer nil
    (format nil "~s"
      (get-selected-text (get-value *current-task* 'sc-viewport)))))

;;; Name: change-slot-content-name-actions
;;; Purpose: if the slot content is a nother schema name, change
;;;          schema name, replace the slot content name in the
;;;          current editing schema, display slot-content-viewport,
;;;          and permitted-slot-content-viewport.
(defun change-slot-content-name-actions (schema slot context)
  (let ((new-name (get-value *current-task* 'current-value))
        (old-name (get-selected-text
                    (get-value *current-task* 'sc-viewport)))
        (slot (get-selected-text
                (get-value *current-task* 's-viewport))))
    (if (schemap old-name)
      (move-schema old-name :to-schema new-name))
    (replace-value (get-value *current-task* 'editing-schema)
       slot old-name :replace new-name)
    (display-slot-contents slot)
    (display-permitted-slot-contents slot)))

;;; Name: get-slot-value-name-action
;;; Purpose: to get the slot value input.
(defun get-slot-value-name-action (schema slot context)
  (get-value-name-action schema slot context)
  (call-method *current-task* 'exit))

;;; Name: delete-slot-content
;;; Purpose: to delete the value from the editing schema, if the
;;;          value is a schema, delete the schema, display the
;;;          slot-content viewport.
(defun delete-slot-content ()
  (let ((slot-name (get-selected-text
                     (get-value *current-task* 's-viewport)))
        (value-name (get-selected-text
```

```
                    (get-value *current-task* 'sc-viewport))))
    (delete-value (get-value *current-task* 'editing-schema)
                  slot-name value-name)
    (if (schemap value-name) (delete-schema value-name))
    (display-slot-contents slot-name)))

;;; Name: edit-slot-schema
;;; Purpose: when the selected content is a schema itself, this
;;;     command  can open another editor windows to edit this
;;;     schema. If the editor for selected content exists, resume
;;;     that editor, otherwise start another set of editor windows.
;;;     If the selected content is not a schema name and the user
;;;     wants to edit it as a schema, then create a schema.
(defun edit-slot-schema ()
  (let* ((schema (get-selected-text
                  (get-value *current-task* 'sc-viewport)))
         (task (taskp schema)))
    (cond (task
           (call-method *current-task* 'pause)
           (call-method task 'resume))
          ((schemap schema)
           (call-method 'dame 'start))
          (t
            (cond ((equal (pop-up-menu 'editing-schema-menu) 't)
                          (create-schema schema)
                          (call-method 'dame 'start))))
    )))

;;; Name: taskp
;;; Purpose: to get the task name of the editor for the given schema.
(defun taskp (schema-name)
  (let ((children (get-values *current-task* 'child-tasks))
        (task nil))
    (dolist (child children task)
      (if (equal (get-value child 'editing-schema) schema-name)
       (setq task child)) )
    task) )

;;; Name: make-editing-schema-menu
;;; Purpose: to make a pop-up menu to ask the user if he/she wants to
;;;      edit the current schema.
(defun make-editing-schema-menu ()
  (make-new-menu 'editing-schema-menu 'pop-up)
  (add-menu-slot 'editing-schema-menu "This value is not a schema."
    :not-selectable t)
  (add-menu-slot 'editing-schema-menu "Do you want to create it?"
    :not-selectable t)
  (add-menu-slot 'editing-schema-menu " " :not-selectable t)
  (add-menu-slot 'editing-schema-menu "Yes"
    :item-action 'value :item-form 't)
  (add-menu-slot 'editing-schema-menu "Not"
    :item-action 'value :item-form 'n)
  (menu-complete 'editing-schema-menu))

(make-editing-schema-menu)
```

```
;;;;;;;;;;;;;;; End slot-content-menu ;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Name: select-permitted-slot-content-predicate
;;; Purpose: to check if the mouse is in the permitted slot contents
;;;              viewport.
(defun select-permitted-slot-content-predicate (schema slot context)
  (cond ((equal (viewport-under-mouse)
                (get-value *current-task* 'psc-viewport))
         t)
        (t nil)))

;;; Name: select-permitted-slot-content-cmd-actions
;;; Purpose: to select an item in the permitted slot contents
;;;     viewport.
(defun select-permitted-slot-content-cmd-actions (schema slot context)
  (select-permitted-content-item))

;;; Name: select-permitted-conten-item
;;; Purpose: to select a permitted slot content.
(defun select-permitted-content-item ()
  (let* ((input-event (call-method 'task-manager 'last-input-event))
         (viewport (viewport-under-mouse))
         (new-item (find-item viewport (input-event-x input-event)
                                       (input-event-y input-event)))
          (item (get-value viewport 'current-item)))
    (if (and new-item
             (or (equal (get-value new-item 'text-name) 'is-a+inv)
                 (equal (get-value new-item 'text-name) 'copy-of+inv)))
      (setq new-item nil))
    (if new-item
      (cond ((equal new-item item) nil)
            (t (if (schemap item)
                   (invert-item-background item viewport))
               (invert-item-background new-item viewport)
               (new-value viewport 'current-item new-item)))
      (beep-viewport viewport))))

;;; Name: permitted-slot-content-menu-cmd-actions
;;; Purpose: to select an item in the permitted slot content viewport,
;;;              and display its related pop-up menu.
(defun permitted-slot-content-menu-cmd-actions (schema slot context)
  (select-permitted-content-item-menu 'permitted-slot-content-menu)
  (call-method 'task-manager 'exit-into *current-task*))

;;; Name: select-permitted-content-item-menu
;;; Purpose: to select a permitted slot content, and display the
;;;     pop-up menu.
(defun select-permitted-content-item-menu (menu-name)
  (let* ((input-event (call-method 'task-manager 'last-input-event))
         (viewport (viewport-under-mouse))
         (item (get-value viewport 'current-item))
         (new-item (find-item viewport (input-event-x input-event)
                                       (input-event-y input-event))))
    (when (and new-item
```

```
            (or (equal (get-value new-item 'text-name) 'is-a+inv)
                (equal (get-value new-item 'text-name) 'copy-of+inv)))
        (setq new-item nil)
        (beep-viewport viewport))
      (when new-item
        (cond ((equal item new-item) nil)
              (t (if (schemap item)
                     (invert-item-background item viewport))
                 (invert-item-background new-item viewport)
                 (new-value viewport 'current-item new-item)))
        (pop-up-menu menu-name))))

;;;;;;;;;;;;;;; Begin permitted-slot-content-menu ;;;;;;;;;;;;;;;

;;; Name: make-permitted-slot-content-menu
;;; Purpose: to make the pop-up menu in the
;;;     permitted-slot-content-viewport.
(defun make-permitted-slot-content-menu ()
  (make-new-menu 'permitted-slot-content-menu 'momentary)
  (add-menu-slot 'permitted-slot-content-menu "Permitted Slot
  Content Menu" :not-selectable t)
  (add-menu-slot 'permitted-slot-content-menu " " :not-selectable t)
  (add-menu-slot 'permitted-slot-content-menu "COPY SCHEMA"
  :item-action 'function :item-form 'copy-permitted-slot-content)
  (add-menu-slot 'permitted-slot-content-menu "ADD VALUE"
  :item-action 'function :item-form 'add-permitted-slot-content)
  (add-menu-slot 'permitted-slot-content-menu "DISPLAY ACTION GRAPH"
  :item-action 'function :item-form 'draw-action-graph)
  (menu-complete 'permitted-slot-content-menu))

(make-permitted-slot-content-menu)

;;; Name: copy-permitted-slot-content
;;; Purpose: tp call copy-permitted-slot-content command.
(defun copy-permitted-slot-content ()
  (call-method 'task-manager 'queue-command
    'copy-permitted-slot-content-cmd))

;;; Name: copy-permitted-slot-content-entry
;;; Purpose: to get the selected text of the permitted slot content.
(defun copy-permitted-slot-content-entry (schema slot context)
  (call-method *current-task* 'set-default 'slot-value-input
    (format nil "~S"
      (symgen (format nil "~S-~S"
                (get-value *current-task* 'hierarchy)
                (get-selected-text
                 (get-value *current-task* 'psc-viewport)))))))

;;; Name: copy-permitted-slot-content-actions
;;; Purpose: to make copy of the selected permitted slot content.
(defun copy-permitted-slot-content-actions (schema slot context)
  (let* ((schema (get-value *current-task* 'editing-schema))
         (schema-to-copy-to (get-value *current-task* 'current-value))
         (slot (get-selected-text
                (get-value *current-task* 's-viewport)))
```

```
        (schema-to-copy-from (get-selected-text
                (get-value *current-task* 'psc-viewport)))
        (value (if (not (equal schema-to-copy-to '||))
                 (duplicate-tree schema-to-copy-from schema-to-copy-to)
                (duplicate-tree schema-to-copy-from))))
    (add-value schema slot value :position :after)
    (display-slot-contents slot)))

;;; Name: add-permitted-slot-content
;;; Purpose: to add the selected permitted slot content directly to
;;;     the editing slot.
(defun add-permitted-slot-content ()
  (let ((schema (get-value *current-task* 'editing-schema))
        (slot (get-selected-text
                (get-value *current-task* 's-viewport)))
        (value (get-selected-text
                 (get-value *current-task* 'psc-viewport))))
    (add-value schema slot value)
    (display-slot-contents slot)))

;;; Functions related to draw-action-graph are listed in the next
;;; appendix.

;;;;;;;;;;;;;;;; End permitted-slot-content-menu ;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;; Begin display-action-graph-menu ;;;;;;;;;;;;;;;;;

;;; Name: display-action-graph-menu-cmd-predicate
;;; Purpose: to check if the mouse is in the
;;;     display-action-graph-viewport.
(defun display-action-graph-menu-cmd-predicate (schema slot context)
  (cond ((member (viewport-under-mouse)
                 (get-values *current-task*
'display-action-graph-viewport)) t)
        (t nil)))

;;; Name: display-action-graph-menu-cmd-actions
;;; Purpose: to display the pop-up menu in the
;;;     display-action-graph-viewport.
(defun display-action-graph-menu-cmd-actions (schema slot context)
  (select-display-action-graph-menu 'display-action-graph-menu)
  (call-method 'task-manager 'exit-into *current-task*))

;;; Name: select-display-action-graph-menu
;;; Purpose: to pop up the command menu.
;;; Input: pop-up-menu name.
(defun select-display-action-graph-menu (menu-name)
  (let* ((input-event (call-method 'task-manager 'last-input-event))
         (viewport (viewport-under-mouse)))
      (pop-up-menu menu-name)
  )
)

;;; Name: make-display-action-graph-menu
;;; Purpose: to make a pop-up menu in the
```

```
;;;     display-action-graph-viewport.
(defun make-display-action-graph-menu ()
  (make-new-menu 'display-action-graph-menu 'momentary)
  (add-menu-slot 'display-action-graph-menu "Display Action
      Graph Menu" :not-selectable t)
  (add-menu-slot 'display-action-graph-menu " " :not-selectable t)
  (add-menu-slot 'display-action-graph-menu "EXIT"
    :item-action 'function :item-form
    'exit-display-action-graph-viewport)
  (menu-complete 'display-action-graph-menu))

(make-display-action-graph-menu)

;;; Name: exit-display-action-graph-viewport
;;; Purpose: to destroy the viewport.
(defun exit-display-action-graph-viewport ()
  (let ((vpt (viewport-under-mouse)))
    (destroy-canvas (get-value
                     (get-value vpt 'on-window)
                     'on-canvas))
    (delete-value *current-task* 'display-action-graph-viewport vpt)))

;;;;;;;;;;;;;; End diplay-action-graph-menu ;;;;;;;;;;;;;;;

;;;;;;;;;;;;;; Begin display-component-menu ;;;;;;;;;;;;;;;

;;; Name: graphic-component-viewport-predicate
;;; Purpose: to predicate if mouse is in display graphic component
;;;     viewport.
(defun graphic-component-viewport-predicate (schema slot cnotext)
  (cond ((equal (viewport-under-mouse)
                (get-value *current-task* 'dsp-viewport))
         (new-values *current-task* 'has-commands
                (get-values 'component-pin-menu-cmd-group 'has-commands)))
        (t nil)))

;;; Name: display-component-menu-cmd-actions
;;; Purpose: to display the pop-up menu in the display graphic
;;;     component veiwport.
(defun display-component-menu-cmd-actions (schema slot context)
  (let* ((input-event (call-method 'task-manager 'last-input-event))
         (vpt (viewport-under-mouse)))
    (pop-up-menu 'display-component-menu))
  (call-method 'task-manager 'exit-into *current-task*))

;;; Name: make-display-component-menu
;;; Purpose: to make the pop-up-menu in the
;;;     display-component-viewport.
(defun make-display-component-menu ()
  (make-new-menu 'display-component-menu 'momentary)
  (add-menu-slot 'display-component-menu "Display Component Menu"
    :not-selectable t)
  (add-menu-slot 'display-component-menu " " :not-selectable t)
  (add-menu-slot 'display-component-menu "BACK"
    :item-action 'function :item-form
```

```
    'back-display-component-viewport)
  (add-menu-slot 'display-component-menu "RESHAPE"
    :item-action 'function :item-form
    'reshape-display-component-viewport)
  (add-menu-slot 'display-component-menu "EXIT"
    :item-action 'function :item-form
    'exit-display-component-viewport)
  (menu-complete 'display-component-menu))

(make-display-component-menu)

;;; Name: back-display-component-viewport
;;; Purpose: to bury the display-component-viewport
(defun back-display-component-viewport ()
  (let ((vpt (viewport-under-mouse)))
    (bury-viewport vpt)))

;;; Name: reshape-display-compoent-viewport
;;; Purpose: to reshape the viewport
(defun reshape-display-component-viewport ()
  (let ((editing-schema (get-value *current-task* 'editing-schema))
        (display-viewport (get-value *current-task* 'dsp-viewport)))
    (destroy-canvas (get-value
                     (get-value display-viewport 'on-window)
                     'on-canvas))
    (create-display-component-viewport)
    (setq display-viewport (get-value *current-task* 'dsp-viewport))
    (display-component editing-schema)
    (link-signals (get-values editing-schema 'has-signal)
       (get-values (get-value (get-value display-viewport 'on-window)
          'on-canvas) 'canvas-items))))

;;; Name: exit-display-component-viewport
;;; Purpose: to exite the display-component-viewport.
(defun exit-display-component-viewport ()
  (let ((vpt (viewport-under-mouse)))
    (destroy-canvas (get-value
                     (get-value vpt 'on-window)
                     'on-canvas))
    (delete-value *current-task* 'dsp-viewport vpt)))

;;;;;;;;;;;;;; End display-component-menu ;;;;;;;;;;;;;;;;;

;;; Name: grph-menu-actions
;;; Purpose: to display a pop-upmenu if the mouse click is on a
;;;     pin in the display graphic component viewport.
(defun grph-menu-actions (schema slot context)
  (if (graphic-pin-item-p)
      (call-method 'pop-up-local-menu 'actions))
  (call-method 'task-manager 'exit-into *current-task*))

;;; Name: graphic-pin-item-p
;;; Purpose: to select graphic pin item
(defun graphic-pin-item-p ()
  (let* ((input-event (call-method 'task-manager 'last-input-event))
```

```
        (viewport (get-value *current-task* 'dsp-viewport))
        (canvas (get-value
                 (get-value viewport 'on-window)
                 'on-canvas))
        (pin-item (get-value *current-task* 'pin-item))
        (item (find-item viewport (input-event-x input-event)
                                  (input-event-y input-event))))
    (when (and (member item (get-values canvas 'canvas-items))
               (member item (browse :simple t :substring "pin")))
      (cond ((equal pin-item nil) nil)
            ((pin-item-rectangle-p pin-item) nil)
            (t (invert-item-background pin-item viewport)
               (new-value pin-item 'background-color nil)))
      (cond ((pin-item-rectangle-p item) nil)
                (t (invert-item-background item viewport)
                (new-value item 'background-color 'invert)))
      (new-value *current-task* 'pin-item item)
      t )))

;;; Name: signal-coconut-actions
;;; Purpose: to open the coconut editor for editing a signal schema.
(defun signal-coconut-actions (schema slot context)
  (let* ((pin-item (get-value *current-task* 'pin-item))
         (component-name (get-value pin-item 'component-name))
         (pin (get-value pin-item 'pin))
         (signal-name (get-value pin-item 'signal-name))
        (signal-schema (make-name component-name signal-name)))
    (cond ((not (equal signal-name nil))
           (call-coconut-fn signal-schema))
          (t (call-method 'task-manager 'queue-command
      'signal-name-cmd)))))

;;;Name: make-name
;;;Purpose: This function makes any name combined by two parts.
(defun make-name (name number)
  (let ((fstr (make-array '(0) :element-type 'string-char
                               :fill-pointer 0
                               :adjustable t)))
   (with-output-to-string (s fstr) (format s "Ã-Ã" name number))
   (read (make-string-input-stream fstr))))

;;; Name: call-coconut-fn
;;; Purpose: to open an coconut editor.
;;; Input: the schema name for editing.
(defun call-coconut-fn (schema-name)
  (new-value 'coconut 'operand-schema schema-name)
  (new-value 'coconut 'reusable-viewports t)
  (call-method 'coconut 'start))

;;; Name: get-signal-name-actions
;;; Purpose: to get the signal name, change pin item from box to
;;;     rectangle, display the signal name beside the pin.
(defun get-signal-name-actions (schema slot context)
  (let* ((signal-name (call-method 'task-manager 'get-input))
```

```
          (pin-item (get-value *current-task* 'pin-item))
          (component-name (get-value pin-item 'component-name))
          (pin (get-value pin-item 'pin))
          (schema-name (make-name component-name signal-name)))
    (cond ((not (equal signal-name nil))
           (cschema schema-name
                 ('copy-of 'signal)
                 ('has-signal+inv component-name)
                 ('name signal-name)
                 ('pin pin))
            (change-pin-item-to-rectangle-fn pin-item)
            (new-value pin-item 'signal-name signal-name)
            (display-signal-name-fn pin-item)
            (call-coconut-fn schema-name))
          (t nil)) )
  (call-method 'task-manager 'exit) )

;;; Name: change-pin-item-to-rectangle-fn
;;; Purpose: to change pin item type to rectangle.
;;; Input: pin-item name.
(defun change-pin-item-to-rectangle-fn (pin-item)
  (let* ((viewport (get-value *current-task* 'dsp-viewport)))
    (delete-any-values *current-task* 'pin-item)
    (cond ((equal (get-value pin-item 'background-color) 'invert)
              (invert-item-background pin-item viewport)
           (new-value pin-item 'background-color nil))
          (t nil))
    (erase-item pin-item viewport)
    (new-value pin-item 'item-type 'rectangle)
    (display-item pin-item viewport)))

;;; Name: delete-signal-exit-actions
;;; Purpose: if delete a signal is confirmed, delete the
;;;     selected signal.
(defun delete-signal-exit-actions (schema slot context)
  (setq s (get-value *current-task* 'delete-symbol))
  (cond ((or (equal s 'y) (equal s 'Y))
         (delete-signal-actions))
        (t nil)))

;;;;;;;;;;;;; Begin delete-signal-actions ;;;;;;;;;;;;;;;;

;;; Name: delete-signal-actions
;;; Purpose: to delete signal schema, change the graphic pin from a
;;;             black bloc to a box.
(defun delete-signal-actions ()
  (let* ((viewport (get-value *current-task* 'dsp-viewport))
         (pin-item (get-value *current-task* 'pin-item))
         (component-name (get-value pin-item 'component-name))
         (signal-name (get-value pin-item 'signal-name))
         (signal-schema (make-name component-name signal-name))
         (item-schema (make-name pin-item 'name)))
    (cond ((schemap signal-schema)
           (delete-schema signal-schema)
           (cond ((schemap item-schema)
```

```
                (erase-item item-schema viewport)
                (destroy-item item-schema))
               (t nil))
          (delete-any-values pin-item 'signal-name)
          (change-pin-item-to-box-fn pin-item))
        (t (pop-up-message* "Signal schema not exists.")) ) ))

;;; Name: change-pin-item-to-box-fn
;;; Purpose: to change the type of the graph pin item from a
;;;     rectangle to a box.
;;; Input: the schema name of the graphic pin item.
(defun change-pin-item-to-box-fn (pin-item)
  (setq viewport (get-value *current-task* 'dsp-viewport))
  (cond ((pin-item-rectangle-p pin-item)
         (erase-item pin-item viewport)
         (new-value pin-item 'item-type 'box)
         (display-item pin-item viewport)
         (invert-item-background pin-item viewport)
         (new-value pin-item 'background-color 'invert))
        (t nil)))

;;; Name: pin-item-rectangle-p
;;; Purpose: to check if the item type is rectangle.
;;; Input: the schema name of a graphic item.
(defun pin-item-rectangle-p (pin-item)
  (cond ((equal (get-value pin-item 'item-type) 'rectangle) t)
        (t nil)))

;;;;;;;;;;;;;; End delete-signal-actions ;;;;;;;;;;;;;;;;;

;;; Name: get-delete-confirm-actions
;;; Purpose: to read the input for confirming delete signal.
(defun get-delete-confirm-actions (schema slot context)
  (get-input-actions 'delete-symbol))

;;; Name: get-input-actions
;;; Purpose: to read the input to the slot in the current task
;;;     schema.
;;; Input: the slot for saving the input in the current task schema.
(defun get-input-actions (task-slot)
       (new-value *current-task* task-slot
              (call-method 'task-manager 'get-input))
       (call-method 'task-manager 'exit))

;;;Name: convert-to-symbol
;;;Purpose: This function is all read-input's convert function.
;;;           It converts input strint into streame.
(defun convert-to-symbol (schema slot context string)
       (values (intern (string-upcase string)
                       (find-package "CRL-USER"))
               t))
```

Appendix D

Functions related to DAME Command Schemata

This appendix lists the lisp code functions related to DAME command schemata.

D.1 Functions related to Save Command Schemata

```
;;; Name: save-icon-actions
;;; Purpose: invert save command icon into black, call save command.
(defun save-icon-actions (schema slot context)
  (invert-icon-to-black schema)
  (call-method *current-task* 'queue-command 'save-cmd))

;Name: inver-icon-to-black
;Purpose: to invert icon color to black.
;Input: icon schema name.
(defun invert-icon-to-black (schema)
  (call-method *current-task* 'place-item-cursor
    (get-value (get-value *current-task* 'command-window)
'icon-viewport)
    (cdr (assoc *current-task* (get-value schema 'graphic-icon)))))

;;; Name: save-cmd-entry
;;; Purpose: to put the editing schema name into the input buffer
;;;     as the default saving file name.
(defun save-cmd-entry (schema slot context)
  (call-method *current-task* 'set-current-input-buffer nil
    (format nil "~S" (get-value *current-task* 'editing-schema))))

;;;;;;;;;;;;;; Begin save-actions ;;;;;;;;;;;;;;;;;;;;;;

;;; Name: save-actions
;;; Purpose: to save the component into the component library.
(defun save-actions (schema slot context)
```

```
  (let ((file-name (get-value *current-task* 'file-name))
        (component (get-value *current-task* 'editing-schema)))
    (cond (file-name
             (new-values 'dame-save-manager 'schemata-list
                 (get-component-tree component (list component)))
           (new-value 'dame-save-manager 'file-to-save-to file-name)
           (call-method 'dame-save-manager 'schemata-save)))
    (call-method *current-task* 'exit)))

;;; Name: get-component-tree
;;; Purpose: to get all schemata of the component.
;;; Input: component name, tree-list is the list of schemata that
;;;     needed to be saved.
(defun get-component-tree (component tree-list)
  (do-local-relational-slots (slot component tree-list)
    (cond ((equal slot 'is-a) nil) ; do not trace is-a relation
          ((equal slot 'copy-of) nil) ; do not trace copy-of relation
          ((equal slot 'instance) nil)
          ; do not trance instance relation
          ((and (position #\+ (format nil "~S" slot))
                (string-equal (format nil "~S" slot) "+inv"
                :start1 (position #\+ (format nil "~S" slot))))
           nil)              ; do not trance any +inv relations
          (t (dolist (value (get-values component slot :path nil))
               (cond ((and (schemap value)
                           (not (member value tree-list)))
                      (push value tree-list)
                      (setq tree-list
                         (get-component-tree value tree-list)))))))
  tree-list))

;;;;;;;;;;;;; End save-actions ;;;;;;;;;;;;;;;;;;

;Name: invert-icon-to-white
;Purpose: to invert icon color to white.
(defun invert-icon-to-white (schema slot context)
  (invert-item-background
    (get-value *current-task* 'item-cursor-item)
    (get-value *current-task* 'item-cursor-viewport))
  (delete-any-values *current-task* 'item-cursor-item)
  (delete-any-values *current-task* 'item-cursor-viewport))

;;; Name: get-save-file-name-action
;;; Purpose: get the input save file name.
(defun get-save-file-name-action (schema slot context)
  (let ((file-name
          (string-downcase
           (format t "~S"
             (call-method 'task-manager 'get-input))))
        (root-schema (get-value *current-task* 'editing-schema)))
    (cond ((and (not (equal (get-value (get-value root-schema
  'copy-of) 'is-a) 'component))
                (equal (pop-up-menu 'save-component-menu) 'n))
           (delete-any-value *current-task* 'file-name))
          (t (cond ((and (probe-file file-name)
```

```
                          (equal (pop-up-menu 'save-schema-menu) 'n))
                     (delete-any-values *current-task* 'file-name))
                    (t (new-value *current-task* 'file-name
 file-name)))))))

;;; Name: make-save-schema-menu
;;; Purpose: to make a pop-up menu for confirming save file when the
;;;       same file name is ready existed.
(defun make-save-schema-menu ()
  (make-new-menu 'save-schema-menu 'pop-up)
  (add-menu-slot 'save-schema-menu "File exists!"
    :not-selectable t)
  (add-menu-slot 'save-schema-menu "Do you want to overwrite?"
    :not-selectable t)
  (add-menu-slot 'save-schema-menu "Yes"
    :item-action 'value :item-form 't)
  (add-menu-slot 'save-schema-menu "No"
    :item-action 'value :item-form 'n)
  (menu-complete 'save-schema-menu)
)

(make-save-schema-menu)

;;; Name: make-save-component-menu
;;; Purpose: to make a pop-up menu for comfirming information.
(defun make-save-component-menu ()
  (make-new-menu 'save-component-menu 'pop-up)
  (add-menu-slot 'save-component-menu "This is not a component!"
    :not-selectable t)
  (add-menu-slot 'save-component-menu "Do you want to save?"
    :not-selectable t)
  (add-menu-slot 'save-component-menu "Yes"
    :item-action 'value :item-form 't)
  (add-menu-slot 'save-component-menu "No"
    :item-action 'value :item-form 'n)
  (menu-complete 'save-component-menu))

(make-save-component-menu)
```

D.2 Functions Related to Load Command Schemata

```
;;; Name: load-icon-actions
;;; Purpose: invert load command icon to black, call load command.
(defun load-icon-actions (schema slot context)
  (invert-icon-to-black schema)
  (call-method *current-task* 'queue-command 'load-cmd))

;;; Name: load-file-action
;;; Purpose: load the file.
(defun load-file-action (schema slot context)
  (let ((file-name (get-value *current-task* 'file-name)))
    (cond (file-name
```

```
          (reload file-name)))
    (call-method *current-task* 'exit)))

;;; Name: reload
;;; Purpose: first delete the schemata with the same names as in
;;;    the file, and load file in.
(defun reload ( filename )
    (with-open-file (file filename :direction :input)
        (do ((input (read file nil) (read file nil)))
            ((not input))
            (when (equal (first input) 'DEFSCHEMA)
                (delete-schema (second input)))))
    (load filename))

;;; Name: get-load-file-name-action
;;; Purpose: get load file name and save it in the file-name slot
;;;    of the *current-task* schema.
(defun get-load-file-name-action (schema slot context)
  (let ((file-name
          (string-downcase
           (format nil "~S"
             (call-method 'task-manager 'get-input)))))
    (cond ((probe-file file-name)
           (new-value *current-task* 'file-name file-name))
          (t (pop-up-message* "No file by this name exists!")
             (delete-any-values *current-task* 'file-name)))))
```

D.3 Functions Related to Create Component Command Schemata

```
;;; Name: create-component-icon-actions
;;; Purpose: if the editor is already editing a schema, exit this
;;;    command; otherwise invert the command icon and call create
;;;    compoment command.
(defun create-component-icon-actions (schema slot context)
  (cond ((get-value *current-task* 'editing-schema)
         (pop-up-message* "Please use a new instance to create the
           component."))
        (t (invert-icon-to-black schema)
           (call-method *current-task* 'queue-command
                 'create-component-cmd))))

;;;;;;;;;;;; Begin create-component ;;;;;;;;;;;;;;;;;;

;;; Name: create-component
;;; Purpose: to create the component, make a copy from the template.
(defun create-component (schema slot context)
  (let ((component-name (get-value *current-task* 'editing-schema)))
    (create-component-nets-fn (get-value *current-task*
'editing-schema)
      (get-value *current-task* 'component-type))
    (new-value component-name 'number-of-pin
      (get-value *current-task* 'number-of-pin)))
```

```
  (replace-base-viewport schema slot context))

;;; Name: create-component-nets-fn
;;; Purpose: make a copy of the component template.
;;; Input: component-name, component-type which is the template name.
(defun create-component-nets-fn (component-name component-type)
  (duplicate-tree component-type component-name)
  (new-value component-name 'package (get-value *current-task*
      'package-name))
  (new-value component-name 'number-of-pin
    (get-value *current-task* 'number-of-pin)))

;;; Name: duplicate-tree
;;; Purpose: to make a copy of the template
;;; Input: parent is the root schema name of a template,
;;;     name is the component name, optional, path is a list of
;;;     relations which you wish to make copies along the
;;;     relations, optional
(defun duplicate-tree (parent &optional (name) (path))
  (when (null name) (setf name (composite-name parent)))
  (copy-schema parent :to-schema name)
  (clear-relations name parent)
  (change-names name parent path)
  name)

;;; Name: clear-relations
;;; Purpose: to delete some slots of the copied schema and add
;;;     copy-of slot into the copied schema.
;;; Input: schema is the name of the copied schema, class is
;;;     the name of the original schema.
(defun clear-relations (schema class)
  (delete-slot schema 'is-a+inv)
  (delete-slot schema 'instance)
  (delete-slot schema 'instance+inv)
  (delete-slot schema 'subclass#)
  (delete-slot schema 'is-a)
  (unless (local-slot-p schema 'copy-of) (create-slot schema
      'copy-of))
  (new-value schema 'copy-of class))

;;; Name: change-names
;;; Purpose: get those relationg whose values are needed to be
;;;     copied, then copy thir values.
;;; Input: schema is the component root schema name,
;;;     parent is the template root name, path is the list of
;;;     relations whose values are needed to be copied.
(defun change-names (schema parent path)
  (dolist (link (get-relation schema path) t)
   (let ((values (get-values schema link)))
       (delete-any-values schema link)
       (dolist (value values)
        (replicate link value schema path)))))

;;; Name: get-relation
;;; Purpose: to get the relations which their values are needed to
```

```
;;;     be copied, also for the relation whose values are not needed
;;;     to be copied, delete its values.
;;; Input: schema is the schema name, path is a list of relations.
;;; Output: a list of relations that their values are needed to be
;;;     copied.
(defun get-relation (schema path)
  (let ((l nil))
   (dolist (slot (get-local-slots schema))
       (when (relationp slot)
        (if path
         (if (member slot path)
          (push slot l)
          (when (not (eq slot 'is-a))
              (delete-any-values schema slot)))
         (when (slotp slot 'copy)
          (if (get-value slot 'copy)
           (push slot l)
           (delete-any-values schema slot))))))
   l))

;;; Name: replicate
;;; Purpose: compose a new name, copy the class schema to the new
;;;     name, clear relations of the copied schema, copy schemata
;;;     along the path recursively.
;;; Input: relation is the link the new schema connecte to the
;;;     parent schema, class is the coresponding schema name in the
;;;     template from which the new schema copied, parent is the
;;;     schema the new schema links to in the component, path is a
;;;     list of relations along which the schamta are copied.
(defun replicate (relation class parent path)
  (let ((name (composite-name class)))
        (copy-schema class :to-schema name)
        (clear-relations name class)
        (change-names name parent path)
        (delete-any-values name (inverse-of relation))
   (add-value parent relation name)))

;;; Name: inverse-of
;;; Purpose: to make the string: relation+inv.
(defun inverse-of (relation)
  (let ((fstr (make-array '(0)
                          :element-type 'string-char
                          :fill-pointer 0
                          :adjustable t)))
   (with-output-to-string (s fstr)
       (format s "Ã+INV" relation))
   (read (make-string-input-stream fstr))))

;;; Name: composite-name
;;; Purpose: to composite a name whose first part is the same
;;;     as the value of class followed by a number.
(defun composite-name (class)
  (make-name class
   (new-value class 'subclass#
```

```
        (1+ (get-value class 'subclass#)))))

;;; Name: replace-base-viewport
;;; Purpose: to display the words on the base viewport top bar.
(defun replace-base-viewport (schema slot context)
  (let ((editing-schema (get-value *current-task* 'editing-schema)))
    (if editing-schema
      (let ((is-a (if (slotp editing-schema 'is-a)
                    (get-value editing-schema 'is-a)
                    nil)))
        (change-viewport-label (get-value *current-task*
    'base-viewport) (get-title editing-schema)))))
  (slot-viewport-redisplay)
  (slot-content-viewport-redisplay)
  (hierarchy-viewport-redisplay)
  (psc-viewport-redisplay))

;;; Name: slot-viewport-redisplay
;;; Purpose: to display the slot viewport.
(defun slot-viewport-redisplay ()
  (let ((viewport (get-value *current-task* 's-viewport))
        (canvas (get-value *current-task* 's-canvas))
        (schema (get-value *current-task* 'editing-schema)))
    (delete-items canvas  viewport)
    (cond ((schemap schema)
           (create-items (sort-relations (get-local-slots schema))
             canvas viewport 1 5)))))

;;; Name: sort-relations,
;;; Purpose: to delete is-a, copy-of, istance, and any invers
;;;     (***+inv) relations from a list of given relations.
(defun sort-relations (list-of-relations)
  (let ((rlist (set-difference list-of-relations
 '(is-a copy-of instance)))
(result nil))
    (dolist (element rlist result)
      (if (and (position #\+ (format nil "~S" element))
       (string-equal (format nil "~S" element) "+inv"
 :start1 (position #\+ (format nil "~S" element))))
(setq result result)
(setq result (append result (list element)))))
    result))

;;; Name: slot-content-viewport-redisplay
;;; Purpose: to display slot-content-viewport.
(defun slot-content-viewport-redisplay ()
  (delete-items (get-value *current-task* 'sc-canvas)
                (get-value *current-task* 'sc-viewport)))

;;; Name: hierarchy-viewport-redisplay
;;; Purpose: to display hierarchy-viewport
(defun hierarchy-viewport-redisplay ()
  (let ((viewport (get-value *current-task* 'h-viewport)))
    (delete-items (get-value *current-task* 'h-canvas) viewport)
    (create-items (append (get-value *current-task* 'hierarchy)
```

```
                    (list (get-value *current-task* 'editing-schema)))
                  (get-value *current-task* 'h-canvas) viewport 1 5)))

;;; Name: psc-viewport-redisplay
;;; Purpose: to display permitted-slot-content-viewport.
(defun psc-viewport-redisplay ()
  (delete-items (get-value *current-task* 'psc-canvas)
                (get-value *current-task* 'psc-viewport)))

;;;;;;;;;;;; End create-component ;;;;;;;;;;;;;;;;;;;;

;;; Name: get-component-name-enty
;;; Purpose: get the component name, if this an existing schema,
;;;     exit the command, otherwise get the component type and
;;;     package name.
(defun get-component-name-entry (schema slot context)
  (get-schema-name-action schema slot context)
  (cond ((schemap (get-value *current-task* 'editing-schema))
            (pop-up-message* "This component exists. Use edit
               schema command to edit it.")
         (delete-any-values *current-task* 'editing-schema)
         (call-method 'task-manager 'quit-into *current-task*))
        (t (new-value *current-task* 'component-type
             (pop-up-menu 'component-type-menu))
           (new-value *current-task* 'package-name
             (pop-up-menu 'component-package-menu)))))

;;; Name: make-component-type-pop-up-menu
;;; Purpose: to list all component types and let the user select
;;;     one type.
(defun make-component-type-pop-up-menu ()
  (make-new-menu 'component-type-menu 'pop-up)
  (add-menu-slot 'component-type-menu "Component type?"
    :not-selectable t)
  (add-menu-slot 'component-type-menu " " :not-selectable t)
  (add-menu-slot 'component-type-menu "Microprocessor"
    :item-action 'value :item-form 'microprocessor)
  (add-menu-slot 'component-type-menu "Memory"
    :item-action 'value :item-form 'memory)
  (menu-complete 'component-type-menu))

(make-component-type-pop-up-menu)

;;; Name: make-package-name-pop-up-menu
;;; Purpose: to list the possible package name, and let the user
;;;     select.
(defun make-package-name-pop-up-menu ()
  (make-new-menu 'component-package-menu 'pop-up)
  (add-menu-slot 'component-package-menu "Package name?"
    :not-selectable t)
  (add-menu-slot 'component-package-menu " " :not-selectable t)
  (add-menu-slot 'component-package-menu "DIP"
    :item-action 'value :item-form 'dip)
  (add-menu-slot 'component-package-menu "G"
    :item-action 'value :item-form 'g)
```

```
  (menu-complete 'component-package-menu))

(make-package-name-pop-up-menu)

;;; Name: get-schema-name-action
;;; Purpose: to get the input string as schema name.
(defun get-schema-name-action (schema slot context)
  (new-value *current-task* 'editing-schema
    (call-method 'task-manager 'get-input)))

;;; Name: get-number-of-pin-actions
;;; Purpose: get the number of pins the component has.
(defun get-number-of-pin-actions (schema slot context)
  (new-value *current-task* 'number-of-pin
    (call-method 'task-manager 'get-input))
  (call-method 'task-manager 'exit-into *current-task*))

;;;Name: convert-to-number
;;;Purpose: This function is the convert function of
;;;     pin-number-input. It converts the input-string into
;;;     number, and it accepts even number only.
(defun convert-to-number (schema slot context string)
       (setq n (with-input-from-string (s string) (read s)))
       (values n (evenp n)))

;;; Name: prompt-error-message
;;; Purpose: pop up a message.
(defun prompt-error-message (schema slot context)
  (pop-up-message* "Please Input Even Number!"))
```

D.4 Functions Related to Edit Schema Command

```
;;; Name: edit-schema-icon-actions
;;; Purpose: to invert icon command, call edit-schema-cmd.
(defun edit-schema-icon-actions (schema slot context)
  (invert-icon-to-black schema)
  (call-method *current-task* 'queue-command 'edit-schema-cmd))

;;; Name: edit-schema-cmd-entry
;;; Purpose: if already in edit command, exit.
(defun edit-schema-cmd-entry (schema slot context)
  (cond ((get-value *current-task* 'editing-schema)
         (pop-up-message* "You are editing a schema!")
        (call-method *current-task* 'exit))))

;;; Name: check-schema-action
;;; Purpose: to if the input string is a schema name.
(defun check-schema-action (schema slot context)
  (if (schemap (get-value *current-task* 'editing-schema))
    (call-method *current-task* 'exit 2)))

;;; Name: create-schema-cmd-actions
```

```
;;; Purpose: if the input string is not a schema name, ask if the
;;;     user wants create a schema, if the answer is yes, create
;;;     the schema, if not quit the command.
(defun create-schema-cmd-actions (schema slot context)
  (cond ((eq (call-method 'task-manager 'get-input) 'y)
         (create-schema (get-value *current-task* 'editing-schema))
         (call-method 'task-manager 'exit-into *current-task*))
        (t
        (delete-any-values *current-task* 'editing-schema)
         (call-method 'task-manager 'quit 2))))
```

D.5 Functions Related to Add-slot Command

```
;;; Name: add-slot-icon-actions
;;; Purpose: to invert command icon, and call add-slot command.
(defun add-slot-icon-actions (schema slot context)
  (invert-icon-to-black schema)
  (call-method *current-task* 'queue-command 'add-slot-cmd))

;;; Name: add-slot-cmd-entry
;;; Purpose: to check if the edit is in the editing schema command.
(defun add-slot-cmd-entry (schema slot context)
  (cond ((get-value *current-task* 'editing-schema) nil)
        (t (pop-up-message* "You are not editing a schema!")
          (call-method *current-task* 'exit))))

;;; Name: get-slot-name-cmd-actions
;;; Purpose: to get the input string as the slot name, create the
;;;     slot if it is a new slot, redisplay the slot viewport.
(defun get-slot-name-cmd-actions (schema slot context)
  (let ((slot (call-method 'task-manager 'get-input))
        (schema (get-value *current-task* 'editing-schema)))
    (cond ((not (local-slot-p schema slot))
           (create-slot schema slot)
           (slot-viewport-redisplay)))
    (call-method 'task-manager 'exit-into *current-task*)))
```

D.6 Functions Related to Add-value Command

```
;;; Name: add-value-icon-actions
;;; Purpose: to invert the command icon, and call add-value command.
(defun add-value-icon-actions (schema slot context)
  (invert-icon-to-black schema)
  (call-method *current-task* 'queue-command 'add-value-cmd))

;;; Name: add-value-cmd-entry
;;; Purpose: if the edit is in editing schema command, get the
;;;     editing slot name, otherwise quit the command.
(defun add-value-cmd-entry (schema slot context)
```

```
  (if (schemap (get-value *current-task* 'editing-schema))
    (let* ((viewport (get-value *current-task* 's-viewport))
           (item (get-value viewport 'current-item)))
      (if item
       (new-value *current-task* 'current-slot
         (get-value item 'text-name))
       (call-method *current-task* 'quit)))
    (call-method *current-task* 'quit)))

;;; Name: add-value-post
;;; Purpose: to add the value into the editing schema, display
;;;     the slot content viewport.
(defun add-value-post (schema slot context)
  (add-value (get-value *current-task* 'editing-schema)
             (get-value *current-task* 'current-slot)
             (get-value *current-task* 'current-value))
  (display-slot-contents slot-name))

;;; Name: get-value-name-action
;;; Purpose: to get the input string as the value.
(defun get-value-name-action (schema slot context)
  (new-value *current-task* 'current-value
    (with-input-from-string (s (call-method 'task-manager
'get-input)) (read s)))
  (call-method *current-task* 'exit))
```

D.7 Functions Related to Return Command

```
;;; Name: return-icon-actions
;;; Purpose: pause the current editing windows, resume the
;;;     parent editing windows.
(defun return-icon-actions (schema slot context)
  (let ((parent (get-value *current-task* 'parent-task)))
    (cond ((equal (get-value parent 'instance) 'dame)
           (call-method  *current-task* 'pause)
           (call-method parent 'resume))
          (t (pop-up-message* "This is the root schema. No where to
                return.")))))
```

D.8 Functions Related to Display Component Command

```
;;; Name: display-component-icon-actions
;;; Purpose: to call display component command.
(defun display-component-icon-actions (schema slot context)
  (call-method *current-task* 'queue-command
      'display-component-cmd))

;;; Name: display-component-actions
```

```
;;; Purpose: if the editing schema is a component root schema,
;;;     display the component, otherwise quit the command; if
;;;     the display viewport does not exist, create the viewport,
;;;     display the component in the viewport, display the signal
;;;     names beside the pins they associate to, otherwise expose
;;;     the existing viewport.
(defun display-component-actions (schema slot context)
  (let ((editing-schema (get-value *current-task* 'editing-schema))
        (display-viewport (get-value *current-task* 'dsp-viewport)))
    (cond ((not (equal editing-schema nil))
           (cond ((equal (get-value (get-value editing-schema 'copy-of)
                                    'is-a) 'component)
                  (cond ((equal display-viewport nil)
                         (create-display-component-viewport)
                         (setq display-viewport
                               (get-value *current-task* 'dsp-viewport))
                         (display-component editing-schema)
                         (link-signals (get-values editing-schema
   'has-signal)
                                       (get-values (get-value
                                           (get-value display-viewport 'on-window)
                                                   'on-canvas) 'canvas-items)))
                        (t (expose-viewport display-viewport))))
                 (t (pop-up-message*
                      (format nil "~S is not a component"
                      editing-schema)))))
          (t (pop-up-message* "No schema!")))))

;;; Name: create-display-component-viewport
;;; Purpose: to create the viewport for displaying the editing
;;;     component graphically.
(defun create-display-component-viewport ()
  (let* ((canvas (create-canvas (symgen 'canvas)))
         (window (create-window (symgen 'window)
                  canvas 0 0 100 100))
         (viewport (create-viewport
                    (cschema (symgen 'viewport) :simple
                      ('viewport-name "component")
                      ('minimum-viewport-width 0.2)
                      ('minimum-viewport-height 0.2)
                      ('has-left-scroll-bar t)
                      ('buried)
                      ('ask-user-for-edges t)) window)))
    (new-value *current-task* 'dsp-viewport viewport)
    (change-window window 0 0
      (* 100 (- (get-value viewport 'xr) (get-value viewport 'xl)))
      (* 100 (- (get-value viewport 'yb) (get-value viewport 'yt))))))

;;; Name: display-component
;;; Purpose: to display the component graphically: to set x, y as
;;;     the cordinates of the top-left point; create the first half
;;;     of the pin items and display them; create the box item and
;;;     display it; create the last half of the pin items and
;;;     display them.
(defun display-component (component-name)
```

```
  (let* ((viewport (get-value *current-task* 'dsp-viewport))
         (window (get-value viewport 'on-window))
         (canvas (get-value window 'on-canvas))
         (pin-number (get-value component-name 'number-of-pin))
         (x (- (/ (+ (get-value window 'xl)
                     (get-value window 'xr))
                  2)
               7))
         (y (+ 5 (get-value window 'yt)))
         (dx 1) (dy 1)
         (width 10)
         (height (* (+ pin-number 1) dy)))
    (new-values *current-task* 'pin-items
      (make-pin-items-fn x y dx dy pin-number 1 canvas viewport
        component-name))
    (make-component-box-fn (+ x dx) (- y dy) dx dy width height
      pin-number canvas viewport component-name
 (get-value component-name 'package))
    (add-values *current-task* 'pin-items
      (make-pin-items-fn (+ (+ x dx) width) y dx dy pin-number
 pin-number canvas viewport component-name))))

;;;Name: make-pin-items-fn
;;;Purpose: This function makes pin buttom icon and display them in
;;;     graphic viewprot.
(defun make-pin-items-fn (x y dx dy number start-number canvas
  viewport component-name)
  (let ((pin-item-list '()))
    (do ((i start-number (cond ((equal start-number number)(1- i))
                               (t (1+ i))))
         (xl x)
         (yt y (+ yt (* 2 dy)) ) )
         ((cond ((equal start-number number)(equal i (/ number 2)))
                (t (equal i (+ 1 (/ number 2))))))
      (let* ((item (create-item (cschema (symgen 'pin) :simple
                                         ('item-type 'box)
                          ('xl xl)('yt yt)
                          ('xr (+ dx xl))('yb (+ dy yt))
                          ('line-thickness 2.0) ('pin i)
                        ('text-x (cond ((equal start-number number)
                                        '+) (t '-) ) )
                          ('signal-name)
                          ('component-name component-name))
                         canvas)))
        (display-item item viewport)
        (push item pin-item-list))) pin-item-list))

;Name: make-component-box-fn
;Purpose: create component box icon
;Input: box's position (xl yt) and size (width height).
;Return: component box icon name.
;Side-effect:
(defun make-component-box-fn (x y dx dy width height pin-number
      canvas viewport component-name
```

```
                                    package-name)
  (let* ((text-item-list (make-text-items-fn x y 0.5 dy pin-number
   1 canvas))
         (cp-box (make-name component-name 'b))
         (cp-name-item (make-name component-name 'n))
         (item-list '()))
    (setq text-item-list (append (make-text-items-fn x y (- width 1)
dy pin-number pin-number canvas) text-item-list))
    (push (create-item
           (cschema 'item-box :simple
              ('item-type 'box)('x1 x)('yt y)
              ('xr (+ x width))('yb (+ y height)))
           canvas) text-item-list)
    (create-icon cp-box canvas x y text-item-list
                 x y (+ x width 1) (+ y height 1) viewport)
    (display-item cp-box viewport)
    (dolist (item text-item-list) (destroy-item item))
    (display-item (create-item (cschema cp-name-item :simple
                               ('item-type 'text-string)
                               ('x x) ('y (+ y height (* 4 dy)))
                               ('text (format nil "~S: ~S-~D"
                                component-name package-name pin-number)) )
                      canvas)
                      viewport)))

;;; Name: make-text-items-fn
;;; Purpose: to create pin number text items for each pin.
;;; Input: x, y: the cordinates,
;;;          dx, dy: the increasement for x and y,
;;;           number: the number of pins,
;;;           start-number: the start number,
;;;           canvas: the canvas name.
(defun make-text-items-fn (x y dx dy number start-number canvas)
  (setq text-item-list '())
  (do ((i start-number (cond ((equal start-number number)(1- i))
                             (t (1+ i))))
       (x1 (cond ((and (equal start-number number) (> start-number 9))
                 (- (+ x dx) 0.5))
                (t (+ x dx)) ) )
       (y1 (+ y (* 2 dy)) (+ y1 (* 2 dy)) ) )
;When the position is out of pk-box-icon's bottom boulder,
; stop the loop.
      ((cond ((equal start-number number)
              (equal i (/ number 2)))
             (t (equal i (+ (/ number 2) 1)))))
    (push (create-item (cschema (make-name 'text i) :simple
                       ('item-type 'text)
                       ('x x1)('y y1)('text (format nil "~D" i))
                       ('font 'tiny-font))
            canvas) text-item-list))
  text-item-list)

;;; Name: link-signals
;;; Purpose: to display signals beside their coresponding pins.
;;; Input: a list of signals, a list of pin items.
```

```
(defun link-signals (signal-list item-list)
  (dolist (signal signal-list)
    (cond ((get-value signal 'pin)
           (dolist (item item-list)
             (if (slotp item 'pin)
               (cond ((equal (get-value signal 'pin)
                             (get-value item 'pin))
                      (change-pin-item-to-rectangle-fn item)
                      (new-value item 'signal-name
                        (get-value signal 'name))
                      (display-signal-name-fn item)))))))))

;;; Name: display-signal-name-fn
;;; Purpose: to display signal names beside the corresponding pin.
;;; Input: a pin item.
(defun display-signal-name-fn (pin-item)
  (let ((signal-name-string (format nil "~S"
    (get-value pin-item 'signal-name)))
       (viewport (get-value *current-task* 'dsp-viewport)) )
    (display-item
      (create-item (cschema (make-name pin-item 'name) :simple
         ('item-type 'text-string)
         ('x (cond ((equal (get-value pin-item 'text-x) '-)
                    (- (get-value pin-item 'xl)
                       (get-string-width-fn
                         signal-name-string
                         'tiny-font viewport)
                       0.5))
                   (t (+ (get-value pin-item 'xr) 0.5)) ) )
          ('y (+ (get-value pin-item 'yt) 1))
          ('text signal-name-string)
          ('font 'tiny-font))
          (get-value pin-item 'in-canvas) )
       viewport) ))

;;; Name: get-string-width-fn
;;; Purpose: to get the strings width.
;;; Input: text-string is the string, font is the font type,
;;;    viewport is the viewport in which the text-string item
;;;    is displayed.
(defun get-string-width-fn (text-string font viewport)
  (let ((l (multiple-value-list
    (bounding-box-for-string  text-string 0 0 font viewport))))
    (- (third l) (first l)) ))
```

D.9 Functions Related to Exit Command Schemata

```
;;; Name: exit-icon-actions
;;; Purpose: to exit current task.
(defun exit-icon-actions (schema slot context)
  (call-method 'task-manager 'exit))
```

Appendix E

Functions Related to Draw-action-graph

```
;;; Name: draw-action-graph
;;; Purpose: if selected permitted slot content is a
;;;    bus-arbitration-protocol, display the action graph.
(defun draw-action-graph ()
  (let* ((protocol (get-selected-text
                    (get-value *current-task* 'psc-viewport)))
         (action-list (if (slotp protocol 'has-action)
                          (get-values protocol 'has-action)
                          nil)))
    (cond ((and (slotp protocol 'is-a)
                (equal (get-value protocol 'is-a)
                 'bus-arbitration-protocol))
           (setq d-graph (get-d-graph action-list))
           (setq nd-graph (get-nd-graph d-graph))
           (setq f-cycle (get-first-cycle d-graph (caar d-graph)))
           (setq f-list (get-faces nd-graph f-cycle))
           (setq graph-mtr
             (get-graph-matrix f-list nd-graph f-cycle (car f-cycle)))
           (create-display-action-graph-viewport)
           (display-action-graph-nodes graph-mtr)
           (display-action-graph-links d-graph))
          (t (pop-up-message* "This is not a protocol."))
    1)))

;;; Name: get-d-graph
;;; Purpose: to get directional graph from actions and
;;;    precedes relation.
(defun get-d-graph (action-list)
  (let ((rslt nil))
    (dolist (action action-list rslt)
      (push (append (list action) (get-values action 'precedes))
  rslt))
    (reverse rslt)))
```

```
;;; Name: get-nd-graph
;;; Purpose: to convert directional graph to non-directional-graph.
(defun get-nd-graph (d-graph)
  (let ((node nil)
        (precedes nil)
        (rslt nil))
    (dolist (node-list d-graph rslt)
      (setq node (car node-list))
      (setq precedes (cdr node-list))
      (dolist (elements (remove node-list d-graph))
       (if (and (member node elements)
                (not (member (car elements) precedes)))
         (setq precedes (append precedes (list (car elements))))))
      (push (append (list node) precedes) rslt))
    (reverse rslt)))

;;; Name: get-first-cycle
;;; Purpose: to get first-cycle from a directional graph.
;;;input: graph list, ((...) (...) ....), each element represents
;;;    a vertes and its neighbores.
;;;output: a list of cycles.
(defun get-first-cycle (directional-graph start-node)
  (cycle-max (findcycles directional-graph (list start-node)
             (cdr (assoc start-node directional-graph)))))

;;; Name: cycle-max
;;; Purpose: to get the longest cycle.
(defun cycle-max (c-list)
  (let ((ln 2)
        (cycle (car c-list)))
    (dolist (c c-list cycle)
      (cond ((> (length c) ln)
             (setq ln (length c))
             (setq cycle c))
            (t nil) ) ) ) )

;;; Name: get-faces
;;; Purpose: to
(defun get-faces (nd-graph first-cycle)
  (let* ((pg-list (genpg first-cycle))
         (f-list (list first-cycle (reverse first-cycle)))
         (npg-list (gennpg nd-graph pg-list))
         (f1 nil)
         (f2 nil))
    (do* ((cmp-list (gencmp nd-graph pg-list)
      (gencmp nd-graph pg-list))
         (cmp-face (chose-cmp-face cmp-list f-list)
              (chose-cmp-face cmp-list f-list)))
      ((not npg-list) f-list)
      (cond ((not cmp-face) (return-from get-faces 'non-planar))
            (t
             (multiple-value-setq (pg-list f1 f2)
               (add-chord (chose-max-chord (car cmp-face) nd-graph)
                 (cadr cmp-face) pg-list))
```

```
            (nsubstitute f1 (cadr cmp-face) f-list :test 'equal)
            (push f2 f-list)
            (setq npg-list (gennpg nd-graph pg-list)))))))

;;; Name: get-graph-matrix
;;; Purpose:
(defun get-graph-matrix (f-list g-list first-face start-node)
  (let* ((sf-list f-list)
         (bondary first-face)
         (sf-list (stable-sort (delete-list sf-list bondary)
                               #'(lambda (x y) (> (length x) (length y)))))
         (mtr (make-initial-mtr (butlast bondary)))
         (jmin 0) (jmax 2)
         (nf-list nil))
    (do ((nf-list (get-uncomplete-faces sf-list bondary)))
        ((null nf-list) mtr)
      (setq sf-list (delete-lists sf-list nf-list))
      (multiple-value-setq (mtr bondary)
       (complete-faces nf-list mtr bondary jmin jmax))
      (cond ((search-matrix mtr 0 (length mtr) 0 0 t 0)
             (setq mtr (add-colmns-matrix 0 mtr 1 '0))
             (setq jmax (1+ jmax))))
      (cond ((search-matrix mtr 0 (length mtr) jmax jmax t 0)
             (setq mtr (add-colmns-matrix (1+ jmax) mtr 1 '0))
             (setq jmax (1+ jmax))))
;       (setq sf-list (delete-list sf-list nf-list))
      (setq nf-list (get-uncomplete-faces sf-list bondary)))))

;;; Name: create-display-action-graph-viewport
;;; Purpose:
(defun create-display-action-graph-viewport ()
  (let* ((canvas (create-canvas (symgen 'canvas)))
         (window (create-window (symgen 'window)
                  canvas 0 0 100 70))
         (viewport (create-viewport
                    (cschema (symgen 'viewport) :simple
                      ('viewport-name "component")
                      ('minimum-viewport-width 0.2)
                      ('minimum-viewport-height 0.2)
                      ('has-left-scroll-bar t)
                      ('buried)
                      ('ask-user-for-edges t)) window)))
    (add-value *current-task* 'display-action-graph-viewport
viewport)
    (change-window window 0 0
      (* 229.6 (- (get-value viewport 'xr) (get-value viewport 'xl)))
      (* 158.2 (- (get-value viewport 'yb)
  (get-value viewport 'yt))))))

;;; Name: display-action-graph-link
;;; Purpose: to display arcs of the action graph.
;;; Input: the directed graph.
(defun display-action-graph-links (directional-graph)
  (let* ((viewport (get-value *current-task*
```

```
      'display-action-graph-viewport))
         (window (get-value viewport 'on-window))
         (canvas (get-value
                  (get-value viewport 'on-window)
                  'on-canvas))
         (gridx (get-value viewport 'gridx))
         (gridy (get-value viewport 'gridy))
         (node0 (get-item canvas 'circle (caar directional-graph)))
         (x0 (get-value node0 'x))
         (dx 10)
         (dy 10))
     (dolist (links directional-graph)                ;dolist1
       (let* ((node1 (get-item canvas 'circle (car links))) ;let2
                (x1 (get-value node1 'x))
              (y1 (get-value node1 'y)))
         (dolist (r (cdr links)) ;dolist2
           (let* ((node2 (get-item canvas 'circle r)) ;let3
                  (x2 (get-value node2 'x))
                  (y2 (get-value node2 'y))
                 (rout (check-rout x0 x1 y1 x2 y2 gridx gridy
  viewport)))
             (cond (rout
                    (setq rout (adjust-rout x0 x1 y1 x2 y2 dx dy
rout))
                    (setq x-arr (make-array (+ 1 (length rout))
                      :initial-contents (cons x1 (mapcar #'car rout))))
                    (setq y-arr
                      (make-array (+ 1 (length rout)) :initial-contents
                                  (cons y1 (mapcar #'cadr rout))))
                    (show-polyline viewport x-arr y-arr
      (+ 1 (length rout)) 'invert 1 'draw)
                    (show-arrow viewport (caar (last rout)) (cadar
      (last rout)) x2 y2 'invert 1 'draw))
                   (t
                    (cond ((> y2 y1)
                           (show-arrow viewport x1 (+ 2 y1) x2 (- y2 2)
                             'invert 1 'draw))
                          ((< y2 y1)
                           (show-arrow viewport x1 (- y1 2) x2 (+ y2 2)
                            'invert 1 'draw))))
             ) ;end cond rout
           ) ;end let3
         ) ;end dolist2
       ) ;end let2
     ) ;end dolist1
   ) ;end let1
)

;;; Name: adjust-rout
;;; Purpose: to adjust the positions of nodes.
;;; Input:
(defun adjust-rout (x0 x1 y1 x2 y2 dx dy rout)
  (cond ((> y2 y1)
         (cond ((= x1 x2)
```

```
                (setq rout (adjust-rout1 2 rout dx)))
                ((and (<= x1 x0) (<= x2 x0))
                (setq rout (adjust-rout1 1 rout dx)))
                ((and (>= x1 x0) (>= x2 x0))
                (setq rout (adjust-rout1 2 rout dx)))))
        ((< y2 y1)
         (cond ((= x1 x2)
                (setq rout (adjust-rout1 1 rout dx)))
                ((> (abs (- x1 x0)) (abs (- x2 x0)))
                (setq rout (adjust-rout1 3 rout dy)))
                ((< (abs (- x1 x0)) (abs (- x2 x0)))
                (setq rout (adjust-rout1 4 rout dy)))))))

;;;Function: adjust-rout1, to change x or y coodinates of a list
;;;     of points.
;;;Input: a point list, ((x1 y1) (x2 y2) ....), type value only
;;;     can be 1, 2, 3, 4. 1 indicates x = x - e; 2: x = x + e;
;;;     3: y = y - e; 4: y = y + e.
;;;Output: a list of points after adjustment
;;;     ((xp1 yp1) (xp2 yp2)....).

(defun adjust-rout1 (type lst e)
  (cond ((= type 1)          ; x - dx
         (mapcar #'(lambda (l) (list (- (car l) e) (cadr l))) lst))
        ((= type 2)          ; x + dx
         (mapcar #'(lambda (l) (list (+ (car l) e) (cadr l))) lst))
        ((= type 3)          ; y - dy
         (mapcar #'(lambda (l) (list (car l) (- (cadr l) e))) lst))
        ((= type 4)          ; y + dy
         (mapcar #'(lambda (l) (list (car l) (+ (cadr l) e))) lst))
        (t (return-from adjust-rout1 "type error"))))


;;;Function: check-rout, find any items on the canvas which is
;;;     overlap with the line between the two points (x1, y1)
;;;     and (x2, y2).
;;;Input: x1, y1, x2, y2, side, gridx, gridy.
;;;Output: a list which contains the points on each of them an
;;;     item is found from point1 to point2.
(defun check-rout (x0 x1 y1 x2 y2 gridx gridy viewport)
  (let ((rslt nil)
        (rslt1 nil)
        (step (if (<= y1 y2) gridy (* -1 gridy)))
        (x 0) (y 0))
    (cond ((>= (abs (- y1 y2)) (* 1.5 gridy))
           (cond ((= x1 x2)
                  (do ((y (+ y1 step) (+ y step)))
                      ((>= (abs (- y y1)) (- (abs (- y2 y1))
                                             (/ gridy 2))))
                    (if (find-item viewport x1 y)
                     (push (list x1 y) rslt)))
                 (setq rslt (reverse rslt)))
                ((not (= x1 x2))
                 (setq rslt
```

```
                    (find-in-rectangle viewport
                      (if (< x1 x2) x1 x2)
                      (if (< y1 y2) y1 y2)
                      (if (> x1 x2) x1 x2)
                      (if (> y1 y2) y1 y2)))
                  (cond (rslt
                         (dolist (item rslt rslt1)
                           (setq y (get-value item 'y))
                           (cond ((not (or (= y y1) (= y y2)))
                                  (setq x (get-value item 'x))
                                  (setq xl (get-line-x x1 y1 x2 y2 y))
                                  (if (or (and (and (>= x1 x0) (>= x2 x0))
                                               (>= x xl))
                                          (and (and (<= x1 x0) (<= x2 x0))
                                               (<= x xl)))
                                    (push (list x y) rslt1)))))
                        (setq rslt
                          (sort rslt1
                                #'(lambda (c1 c2)
                                    (< (abs (- (cadr c1) y1))
                                      (abs (- (cadr c2) y1))))))))))
            rslt)
          (t nil))))

(defun get-line-x (x1 y1 x2 y2 y)
  (let ((a (if (not (= y1 y2))
               (/ (- x2 x1) (- y2 y1))
               'inf)))
    (if (equal a 'inf) x1 (+ (* a (- y y1)) x1))))

;;; Name: get-item
;;; Purpose: given item type and the node name, to get the item name.
;;; Input, canvas name, item-type, and node-name.
(defun get-item (canvas item-type node-name)
  (let ((canvas-items (get-values canvas 'canvas-items)))
    (dolist (item canvas-items)
      (if (and (equal (get-value item 'item-type) item-type)
               (equal (get-value item 'node-name) node-name))
        (return-from get-item item)))))

;;; Name: display-action-graph-nodes
;;; Purpose: to display the nodes of an action graph.
;;; Input: the node position matrix.
(defun display-action-graph-nodes (graph-mtr)
  (let* ((viewport (get-value *current-task*
     'display-action-graph-viewport))
        (window (get-value viewport 'on-window))
        (canvas (get-value
                 (get-value viewport 'on-window)
                 'on-canvas))
         (xl (get-value window 'xl))
          (xr (get-value window 'xr))
          (yt (get-value window 'yt))
          (yb (get-value window 'yb))
        (scalex 0.8)
```

```
        (scaley 0.8)
        (gridx (* (/ (- xr xl) (length (car graph-mtr))) scalex))
        (gridy (* (/ (- yb yt) (length graph-mtr)) scaley))
        (i 0) (j 0) (x 0) (y 0))
    (create-slot viewport 'gridx)
    (new-value viewport 'gridx gridx)
    (create-slot viewport 'gridy)
    (new-value viewport 'gridy gridy)
    (dolist (row graph-mtr)
      (setq i (1+ i))
      (setq j 0)
      (dolist (node row)
       (setq j (1+ j))
       (cond ((not (equal node '0))
              (setq x (* j gridx))
              (setq y (* i gridy))
              (display-item (create-item (cschema (unique-name)
                                                  :simple
                                          ('item-type 'circle)
                                          ('x x) ('y y)
                                          ('radius 2)
                                          ('node-name node))
                                 canvas) viewport)
              (show-string viewport (+ x 3) y
                (format nil "~S" (get-value node 'function))
                'invert 'draw 'screen-12)))))))

;;; Name: findcycles,
;;; Purpose: to find cycles.
;;;Input: graph-list (g-list), patial-cycle-list (p-list), and
;;;          neibore-nodes(nb-list)
;;;Output: a list consists all the cycles.
(defun findcycles (g-list p-list nb-list)
  (let ((c-list nil))
    (dolist (node nb-list c-list)
      (cond ((equal node (car p-list))
                (push (append p-list (list node)) c-list))
            ((member node p-list) nil)
            (t (setq  c-list (append c-list
                 (findcycles g-list (append p-list (list node))
                   (get-neighbores g-list node (car (last p-list)))))))))
      ) ) ) )

;;; Name: get-neighbores
;;; Purpose: to get the neighboring nodes of the given node.
;;; Input: g-list is a graph, node is a node in the graph, prenode
;;;     is the node preeds the given node.
(defun get-neighbores (g-list node prenode)
  (set-difference (assoc node g-list) (list node prenode)
      :test 'equal))

;;; Name: genpg
;;; Purpose: generate a partial graph
;;; Input: a cycle, (1 2 7 3 8 9 4 5 6 1)
```

```
;;; Output: ((1 2 6) (2 1 7) (7 2 3) (3 7 8) ....)
(defun genpg (cycle)
  (let ((pg nil)
(invc (list-reverse cycle)))
    (dolist (node cycle pg)
      (push
(list node (cadr (member node cycle)) (cadr (member node invc)))
pg))
    (list-reverse (cdr pg)) ) )

;;; Name: gennpg
;;; Purpose: to generate npg, npg = g - pg.
;;; Input: g-list pg-list
;;; Output: npg-list
(defun gennpg (g-list pg-list)
  (let ((e-list nil))
    (dolist (vertex g-list e-list)
      (let ((node (car vertex)))
(if (and (assoc node pg-list)
 (set-difference vertex (assoc node pg-list)))
  (push (cons node (set-difference vertex (assoc node pg-list)))
    e-list)) ) )
    (list-reverse e-list) ) )

;;; Name: gencmp
;;; Purpose: to generate components
;;; Input: graph and partial graph
;;; Output: components list. A component is represented by a
;;;     two-element list ((...)(...)), the first element
;;;     represents the nodes which are on the partial graph,
;;;     the second element represents the nodes which are
;;;     not on the partial graph.
(defun gencmp (g-list pg-list)
  (let ((npg-list (gennpg g-list pg-list))
(cmp-list nil))
    (do* ((vn-list (pop npg-list) (pop npg-list))
  (vertex (pop vn-list) (pop vn-list))
  (neibore (pop vn-list) (pop vn-list))
  (cmp (list (list vertex) '())
       (list (list vertex) '())))
 ((not (car npg-list)) cmp-list)
 (setq npg-list
   (if vn-list (cons (cons vertex vn-list) npg-list)
npg-list))
         (cond ((assoc neibore pg-list)  ;simple component
          (push (list (cons neibore (car cmp)) (cadr cmp))
       cmp-list)
          (setq npg-list (update-npg-list vertex neibore npg-list)) )
         (t ;complex component
(cond ((exploredp-n neibore cmp-list) nil)
       (t (setq results
    (explore-vertex vertex (list neibore)
   g-list pg-list npg-list cmp))
              (setq cmp-list (cons (car results) cmp-list))
              (setq npg-list (cadr results)))
```

```
)) ) ) ) )

;;; Name: explore-vertex
;;; Purpose: to explore a vertex which is not on the partial
;;;      graph, and find the component.
;;; Input: the vertex, its neighbores, the partial graph(pg-list),
;;;    npg-list.
;;; Output: the component list((end points)(vertices)), update the
;;;      npg-list.
(defun explore-vertex (vertex neighbores g-list pg-list npg-list cmp)
  (let ((component cmp)
(updated-npg-list npg-list)
(results nil))
    (dolist (n neighbores)
      (cond ((exploredp n component) nil)
    ((assoc n pg-list)
     (setq component
       (list (cons n (car component)) (cadr component))))
    (t (setq component
 (list (car component) (cons n (cadr component))))
       (setq updated-npg-list
 (update-npg-list vertex n  updated-npg-list))
       (setq results
 (explore-vertex n (get-neighbores g-list n vertex) g-list
   pg-list updated-npg-list component))
       (setq component (car results))
       (setq updated-npg-list (cadr results)) ) ) )
    (list component updated-npg-list) ) )

;;; Name: update-npg-list
;;; Purpose: to update the ~partial-graph list
;;; Input: the vertex which is under exploring and its exploring
;;;     neighbore, and the npg-list.
;;; Output: the updated npg-list.
(defun update-npg-list (vertex neibore npg-list)
  (let ((result nil))
    (dolist (n-list npg-list result)
      (if (equal (car n-list) neibore)
(if (del-el vertex (cdr n-list))
          (push
    (cons (car n-list) (del-el vertex (cdr n-list)))
    result))
(push n-list result)) )
    (list-reverse result) ) )

;;; Name: del-el
;;; Purpose: to delete the given element from a list.
(defun del-el (element list)
  (let ((result nil))
    (dolist (n list result)
      (if (not (equal n element))
(push n result)) )
    (list-reverse result) ) )

;;; Name: exploredp
```

```
;;; Purpose: to check if the given node n is already on the component.
(defun exploredp (n component)
  (if (or (member n (car component)) (member n (cadr component)))
t nil) )

;;; Name: exploredp-n
;;; Purpose: to check if the given node n is in an component.
;;; Input: n is the node, cmp-list is a list of components.
(defun exploredp-n (n cmp-list)
  (let ((result nil))
    (dolist (cmp cmp-list result)
      (if (member n (cadr cmp)) (setq result t))) ) )

;;; Name: complete-faces
;;; Purpose: to complete the non-complete faces on the current
;;;    bondary. There are two cases:
;;;    1. If the non-complete face has no extra node other than
;;;    the current bondary, then just change the current bondary.
;;;    2. If the non-complete face has exta nodes other than the
;;;    current bondary, then add these extra nodes, change the
;;; current bondary.
;;; Input: a list contains non-complete-faces (nf-list), a matrix
;;;    (mtr), the current bondary (bondary), j left and j right
;;;    (jl, jr).
;;; Output: an updated matrix, an updated bondary, updated j left
;;;    and jright.
(defun complete-faces (nf-list mtr bondary jl jr)
  (let ((upmtr mtr)
(upbond bondary))
    (dolist (f nf-list upmtr)
      (cond ((and (subsetp f upbond) (subsetp upbond f)) nil)
    ((subsetp f upbond)
     (setq subbond (intersection (butlast upbond) f))
     (setq ns (car subbond) ne (car (last subbond)))
     (if (location-rightp ns ne f)
       (setq upmtr (adjust-nodes ns ne 'right upmtr jl jr))
       (setq upmtr (adjust-nodes ns ne 'left upmtr jl jr)) )
     (delete-if #'atom upbond :start (1+ (position ns upbond))
      :end (position ne upbond)) )
    (t (setq subbond (get-subbondary f upbond))
      (setq upmtr (add-new-nodes subbond upmtr jl jr))
      (setq ns (car subbond) ne (car (last subbond)))
      (setq upbond
        (if (= (position ne upbond) 0)
  (append (subseq upbond 0 (position ns upbond)) subbond)
  (append (subseq upbond 0 (position ns upbond))
  subbond
  (subseq upbond (1+ (position ne upbond))))))
    ) ) )
    (values upmtr upbond) ) )

;;; Name: adjust-nodes
;;; Purpose: if a extra face is on a current bondary,  a line will
;;;    draw between the two nodes which are are already exist;
```

```
;;;     adust-nodes is used to check if this extra line could be
;;;     drawn between the two nodes without crossing any other lines.
;;;     If yes, adjust the location of these nodes.
;;; Input: two nodes on the bondary, ns: start node, ne: end node,
;;;     and the location side in the matrix which is either 'right or
;;;     'left (side), and the matrix and the position of colmn (j).
;;; Output: a updated matrix and jmin or jmax (if necessary).
(defun adjust-nodes (ns ne side mtr jl jr)
  (let* ((p1 (get-position ns mtr))
 (p2 (get-position ne mtr))
 (i1 (if (< (car p1) (car p2)) (car p1) (car p2)))
 (i2 (if (= i1 (car p1)) (car p2) (car p1)))
 (j1 (if (= i1 (car p1)) (cadr p1) (cadr p2)))
 (j2 (if (= j1 (cadr p1)) (cadr p2) (cadr p1)))
 (j (if (equal side 'right)
(if (<= j1 j2) j1 j2)
(if (>= j1 j2) j1 j2)))
 (exist-nodes (if (> (- i2 i1) 1)
(search-matrix mtr i1 i2 j1 j2 side
  (if (equal side 'right) jr jl))
nil))
 (jmax -1)
 (jmin -1)
 (upmtr mtr))
    (cond (exist-nodes
   (if (equal side 'right)
     (setq jmax
       (car (sort (mapcar #'third exist-nodes)
      #'(lambda (x y) (> x y))))
   j (1+ jmax))
     (setq jmin
       (car (sort (mapcar #'third exist-nodes)
      #'(lambda (x y) (< x y))))
   j (if (= jmin 0) 0 (- jmin 1))) )
   (cond ((not (= jmax -1))
  (setq j (1+ jmax)))
 ((not (= jmin -1))
  (setq j (- jmin 1))) )
   (cond ((and (= j1 1) (= j2 1))
  nil)
 ((= j1 1)
  (if (equal side 'right)
    (setq upmtr (move-item-in-matrix upmtr
  i2 j2 i1 (if (< j2 j) j j2)))
    (setq upmtr (move-item-in-matrix upmtr
  i2 j2 i1 (if (> j2 j) j j2)))))
 ((= j2 1)
  (if (equal side 'right)
    (setq upmtr (move-item-in-matrix upmtr
  i1 j1 i1 (if (< j1 j) j j1)))
    (setq upmtr (move-item-in-matrix upmtr
  i1 j1 i1 (if (> j1 j) j j1)))) )
```

```
 (t (if (equal side 'right)
                    (setq upmtr (move-item-in-matrix upmtr
  i1 j1 i1 (if (< j1 j) j j1)))
    (setq upmtr (move-item-in-matrix upmtr
  i1 j1 i1 (if (> j1 j) j j1))))
  (if (equal side 'right)
                    (setq upmtr (move-item-in-matrix upmtr
  i2 j2 i2 (if (< j2 j) j j2)))
    (setq upmtr (move-item-in-matrix upmtr
  i2 j2 i2 (if (> j2 j) j j2))))) ))
  (t nil) )
    upmtr ) )

;;; Name: move-item-in-matrix
;;; Purpose: to move an item at position i1 j1 to position
;;;       i2 j2, and replace the element at i1 j1 with '0.
;;; Input: a matrix (mtr), position1 (i1 j1), and position2 (i2 j2).
;;; Output: a matrix.
(defun move-item-in-matrix (mtr i1 j1 i2 j2)
  (let ((upmtr mtr)
(n (nth j1 (nth i1 mtr))))
    (cond ((and (= i1 i2) (= j1 j2)) upmtr)
  (t (setq upmtr (replace-item-in-matrix upmtr n i2 j2))
             (setq upmtr (replace-item-in-matrix upmtr '0 i1 j1))))))

;;; Name: replace-item-in-matrix
;;; Purpose: to replace an element in the matrix at the
;;;       given position (i j) with the given item.
;;;Input: the new item (item), and the position i and j.
;;;Output: the replaced matrix.
(defun replace-item-in-matrix (mtr item i j)
  (let ((upmtr mtr))
    (replace upmtr
      (list (replace (nth i upmtr)
     (list item)
     :start1 j :end1 (1+ j)))
      :start1 i :end1 (1+ i)) ) )

;;; Name: search-matrix
;;; Purpose: to find any element in a matris which is not '0 in
;;;       the given area by i1 i2 j1 j2 and side.
;;;Input: a mtrix (mtr), from row i1 to i2, from colmn j1 to j2.
;;;Output: a list which contains element and their positions,
;;;     ((n i j) ...).
(defun search-matrix (mtr i1 i2 j1 j2 side js)
  (let ((rslt nil)
(j0 (if (equal side 'left)
js
        (if (<= j1 j2) j1 j2)))
(je (if (equal side 'right)
js
(if (<= j1 j2) j2 j1)))
```

```
(a (if (not (= i1 i2))
       (/ (- j2 j1) (- i2 i1))
       'inf))
(x 0))
     (do ((i (1+ i1) (1+ i)))
         ((= i i2))
       (do ((r (subseq (nth i mtr) j0 (1+ je)))
    (n nil)
    (j j0 (1+ j)))
   ((= j (1+ je)))
(setq y (if (equal a 'inf) j
  (+ (* a (- i i1)) j1)))
(setq n (pop r))
(cond ((equal side 'right)
        (if (and (>= j y) (not (equal n '0)))
     (push (list n i j) rslt)))
       ((equal side 'left)
        (if (and (<= j y) (not (equal n '0)))
     (push (list n i j) rslt)))
       (t (if (not (equal n '0))
     (push (list n i j) rslt))) ) ) )
     rslt ) )

;;; Predicate: location-rightp
;;; Purpose: to find if the face should be allocated at the
;;;      right side or the left side of the matrix.
;;;      If the order of the nodes on the face is start node
;;;      followed by the end node, according to right hand side
;;;      rule, the location should be on the right side.
;;; Input: a start node on the bondary (ns), and an end node on the
;;;    bondary (ne).
;;; Output: If the location is at the right side of the matrix,
;;;      return t, otherwise nil.
(defun location-rightp (ns ne face)
  (let ((f (butlast face)))
    (do ()
((equal (car f) ns))
      (if (not (equal (car f) ns))
(setq f (append (cdr f) (list (car f))))) )
     (if (equal (cadr f) ne)
      t nil) ) )

;;; Name: add-new-nodes
;;; Purpose: to given a list of nodes as subbondary that is the
;;;      ends of the list are on the current bondary.
;;;      By finding the positions of the end nodes in the matrix,
;;;      the location the face can be fond, because the right
;;;      hand rule for bondary (the inclosed eare always at the
;;;      right hand side of the face).
;;;      1. if the positions of the first node and the last node
;;; is up and low in the matrix, then the location of the
;;; subbondary is at the right side of the matrix.
;;;      2. otherwise, the location is at the left side of the matrix.
;;; Input: a list of nodes which the first and the last nodes are on
```

```
;;;   the current bondary (subbond), a matrix, j left and j right
;;;   (jl, jr).
;;; Output: an updated matrix, updated j left, updated j right.
(defun add-new-nodes (subbond mtr jl jr)
  (let* ((ns (car subbond))
 (ne (car (last subbond)))
 (pns (get-position ns mtr))
 (pne (get-position ne mtr))
 (j nil)
 (side nil)
 (upmtr mtr))
    (cond ((< (car pns) (car pne))
   (setq side 'right)
   (setq j jr)
   (setq i1 (car pns) i2 (car pne)) )
  ((> (car pns) (car pne))
   (setq side 'left)
   (setq i1 (car pne) i2 (car pns))
   (setq j jl) )
  (t (return-from add-new-nodes 'error)) )
    (setq in (- (- i2 i1) 1))
    (setq xn (butlast (cdr subbond)) s (length xn))
    (cond ((< in s)
   (setq upmtr (add-rows-matrix (1+ i1) upmtr (- s in) '0))
   (setq i2 (+ i2 (- s in)))
   (setq in (1- (- i2 i1))) )
  (t nil) )
    (setq xn (get-xnodes-positions xn in i1 j))
    (setq upmtr (add-nodes-matrix xn upmtr))
    (cond ((equal side 'right)
   (setq upmtr
     (adjust-nodes ns (caar xn) 'right upmtr jl jr))
   (setq upmtr
     (adjust-nodes ne (caar (last xn)) 'right upmtr jl jr)))
  ((equal side 'left)
   (setq upmtr
     (adjust-nodes ne (caar xn) 'left upmtr jl jr))
   (setq upmtr
     (adjust-nodes ns (caar (last xn)) 'left upmtr jl jr))))
    upmtr ) )

;;; Name: add-nodes-matrix
;;; Purpose: to add nodes into a matrix.
;;; Input: a sequence of lists, ((node positioni positionj)....),
;;;    (np-list), and a matrix (mtr).
;;; Output: an updated matrix.
(defun add-nodes-matrix (np-list mtr)
  (let ((rslt mtr))
    (dolist (np np-list rslt)
      (setq r (nth (second np) rslt))
      (replace rslt
       (list (replace r (list (car np)) :start1 (third np)) )
       :start1 (second np)) ) ) )
```

```
;;; Name: get-xnodes-positions
;;; Purpose: give a list of nodes (xn), to get the positions of
;;;    each node in a matrix.
;;; Input: xn is a list of nodes, in is the number of rows for
;;;    the list, i1 is the starting row, j is the colmn.
;;; Outpur: a list, each element of a list is tuple: (node row colmn).
(defun get-xnodes-positions (xn in i1 j)
  (let ((k (floor (/ in (1+ (length xn)))))
(rslt nil)
(i 0))
    (if (= k 0) (setq k 1))
    (dolist (x xn rslt)
      (setq i (1+ i))
      (push (list x (+ i1 (* k i)) j) rslt) )
    (reverse rslt) ) )

;;; Name: add-rows-matrix
;;; Purpose: to add a row in a matrix.
;;; Input: positioni is the starting row, mtr is the matrix,
;;;     number-of-rows is the number of rows to be added,
;;;    sym is the value of each element of the added rows.
(defun add-rows-matrix (positioni mtr number-of-rows sym)
  (let ((rslt mtr))
    (dotimes (x number-of-rows rslt)
      (setq rslt (append (subseq rslt 0 positioni)
   (list (make-list (length (car rslt)) :initial-element sym))
   (subseq rslt positioni))) ) ) )

;;; Name: add-colmns-matrix
;;; Purpose: to add colmns in a matrix.
;;; Input: pj is the starting colmn, mtr is the matrix,
;;;    number-of-colmns is the number of colmns to be added, sym
;;;    is the value of each element of the added colmns.
(defun add-colmns-matrix (pj mtr number-of-colmns sym)
  (let ((nmtr mtr)
(tmp nil))
    (dotimes (x number-of-colmns nmtr)
      (dolist (r nmtr tmp)
(push (append (subseq r 0 pj) (list sym) (subseq r pj)) tmp) )
      (setq nmtr (reverse tmp))
      (setq tmp nil) ) ) )

;;; Name: get-subbondary
;;; Purpose: to arrange the subbondary as (ns x1 x2 .. xs ne) which
;;;      ns and ne are the end nodes on the bondary.
;;; Input: a uncompleted face which has at lease to nodes on the
;;;    bondary, and the bondary.
;;; Output: a list which contains the two end nodes on the bondary
;;;    and other nodes which are not onthe current bondary.
(defun get-subbondary (face bondary)
  (let* ((subbond (intersection (butlast bondary) (butlast face)))
 (ns (car subbond))
 (ne (car (last subbond)))
 (sub nil) )
```

```
    (setq subbond
      (do ((sub (butlast face))
   (n1 (car sub) (car sub))
   (n2 (cadr sub) (cadr sub)))
  ((and (member n1 bondary) (not (member n2 bondary))) sub)
       (setq sub (append (cdr sub) (list (car sub)))) ) )
    (setq subbond
      (do ((i 1)
   (n1 (car subbond))
   (n2 (cadr subbond)))
   ((and (not (member n1 bondary)) (member n2 bondary))
    (subseq subbond 0 (1+ i)))
(setq i (1+ i))
(setq n1 n2)
(setq n2 (nth i subbond)) ) )
    subbond ) )

;;; Name: get-uncomplete-faces
;;; Purpose: if a face has at least 2 nodes intersect with the
;;;    current bondary, them this face is a uncomplete-face.
;;; Input: a list of faces (sf-list), and the bondary.
;;; Output: a list of uncomplete faces.
(defun get-uncomplete-faces (sf-list bondary)
  (let ((nf-list nil)
   (bb (butlast bondary))
(b_f nil))
    (dolist (f sf-list nf-list)
      (setq b_f (intersection bb (butlast f)))
      (cond ((equal (list (first f) (second f))
    (list (first bb) (car (last bb))))
     nil)
    ((> (length b_f) 1)
     (setq subbb (subseq bb (position (car b_f) bb)
             (1+ (position (car (last b_f)) bb)) ) )
     (if (= (length b_f) (length subbb))
       (push f nf-list)) ) ) ) ) )

;;; Name: delete-lists
;;; Purpose: to delete seq2 from seq1.
;;; Input: sequence1, sequance2, their elements are lists,
;;;    (lst1 lst2 ....).
;;; Output: sequence1 - sequence2, if the element set in seq2 is
;;;    equal to the element set in seq1, i.e. let
;;;    seq1 = (l1 l2 .... li ... ln), and
;;;    seq2 = (p1 p2 ... pj ... pm), if set(pj) = set(li), then
;;;    seq1 = seq1 - (li).
(defun delete-lists (seq1 seq2)
  (let ((rslt seq1))
    (dolist (f seq2 rslt)
      (setq rslt (delete-list rslt f)) ) ) )

;;; Name: delete-list
;;; Purpose: to delete a list frome a sequence if the set of
;;;    elements in the list is equal to the set of element in
```

```
;;;     one of the list in the sequence.
;;; Input: a sequence which contains list elements (seq),
;;;    (list1 list2 ...), and a list which contains atom
;;;     elements (lst), (atom1 atom2 ...).
;;; Output: an updated sequence which equals seq - (lst), iff lst
;;;     as a set is equal to one of the list as a set in seq.
(defun delete-list (seq lst)
  (remove-if #'(lambda (x) (and (subsetp (remove-duplicates x)
 (remove-duplicates lst))
(subsetp (remove-duplicates lst)
 (remove-duplicates x))))
    seq) )

;;; Name: get-position
;;; Purpose: to get a node's position index from a matrix.
;;; Input: a matrix (mtr), and a node.
;;; Output: a cons pair contains the position index,
;;;     i.e. (row colmn).
(defun get-position (node mtr)
  (let ((i -1)
(j 0))
    (dolist (row mtr)
      (setq i (1+ i) j -1)
      (dolist (n row)
(setq j (1+ j))
(if (equal node n)
  (return-from get-position (list i j))) ) ) ) )

;;; Name: make-initial-mtr
;;; Purpose: to make the initial matrix from a list.
;;; Input: a list.
(defun make-initial-mtr (lst)
  (append (list (list '0 (car lst) '0))
(mapcar #'(lambda (x) (list '0 x '0)) (cdr lst))) )

;;; Name: chose-cmp-face
;;; Purpose: to choose a component that can fit into one face.
;;; Input: cmp-list is a list of components, f-list is a list
;;;     of faces.
(defun chose-cmp-face (cmp-list f-list)
  (let ((faces nil)
(result nil))
    (dolist (cmp cmp-list)
      (setq faces (test-cmp cmp f-list))
      (cond ((not faces)
       (return-from chose-cmp-face nil))
    (t (push (cons cmp faces) result))))
    (setq result
  (sort result #'(lambda (x y) (<= (length x) (length y)))))
    (list (caar result) (cadar result))))

;;; Name: test-cmp
;;; Purpose: to check if the component can be fitted in a face.
;;; Input: cmp is a component, f-list is a list of faces.
```

```
(defun test-cmp (cmp f-list)
  (let ((result nil))
    (dolist (f f-list result)
      (if (set-difference (car cmp) f) nil
(push f result)))))

;;; Name:: chose-max-chord
;;; Purpose: to choose a chord which has maximum length.
;;; Input: the component (cmp), the graph (g-list).
;;; Output: a chord which has the maximum length.
(defun chose-max-chord (cmp g-list)
  (let ((chords nil))
    (if (cadr cmp)
      (setq chords (chose-chords g-list cmp (list (caar cmp))
  chords))
      (setq chords (list (car cmp))))
    (car (sort chords #'(lambda (x y) (> (length x) (length y)))))))

;;; Name: chose-chords
;;; Purpose: to find a set of chords from a component.
;;; Input: g-list is the graph, cmp is the component, pchord is
;;;     a partial chord in the component, chords are the chords
;;;     that is already found.
(defun chose-chords (g-list cmp pchord chords)
  (let* ((upchords chords)
 (exterio (set-difference (cadr cmp) (cdr pchord)))
 (interio (remove (car pchord) (car cmp)))
 (neibores (cdr (assoc (car (last pchord)) g-list)))
 (explore-e (intersection exterio neibores))
 (explore-i (intersection interio neibores)))
    (if explore-i
      (push (append pchord (list (car explore-i))) upchords))
    (dolist (v explore-e upchords)
      (setq upchords (chose-chords g-list cmp
(append pchord (list v)) upchords)) )))

;;; Name: add-chord
;;; Purpose: to add a chord to a face.
;;; Input: chord is the chord to be added, face is the face that the
;;;     chord to be put in, pg-list is the partial graph that has
;;;     been explored.
(defun add-chord (chord face pg-list)
  (let ((upg-list nil)
(f1 nil)
(f2 nil))
    (setq upg-list (update-pg-list chord pg-list))
    (setq upg-list (update-pg-list (reverse chord) upg-list))
    (multiple-value-setq (f1 f2) (update-face chord face))
    (values upg-list f1 f2)))

;;; Name:: update-face
;;; Purpose: to add one chord to a face, split the face into two.
;;;       for example, face (1 2 3 4 5 6 1) and chord (1 8 9 2), into
;;;       face1 (1 8 9 2 3 4 5 6 1) and face2 (2 9 8 1 2).
;;; Input: the chord and the face.
```

```
;;; Output: the two new faces.
(defun update-face (chord face)
  (let* ((c (if (> (position (car chord) face)
    (position (car (last chord)) face))
     (reverse chord) chord))
 (s1 (subseq face 0 (position (car c) face)))
 (s2 (subseq face (position (car c) face)
  (1+ (position (car (last c)) face))))
 (s3 (subseq face (1+ (position (car (last c)) face))))
 (f1 (append s1 c s3))
 (f2 (append (reverse c) (cdr s2))))
    (values f1 f2)))

;;; Name:: update-pg-list
;;; Purpose: add the chord into the partial graph.
;;; Input: the chord, and the partial graph (pg-list).
;;; Output: the updated partial graph.
(defun update-pg-list (chord pg-list)
  (let ((result pg-list))
    (do ((v (pop chord) (pop chord))
 (n (car chord) (car chord)))
 ((not n) result)
      (cond ((assoc v pg-list)
     (nsubstitute (append (assoc v result) (list n))
                  (assoc v result)
 result
         :test #'equal))
    (t (setq result (append result (list (list v n)))))))))
```

Vita

Surname:	Li	Given Names:	Dongni
Place of Birth:	Shanghai, P.R. China	Date of Birth:	1960-10-28

Educational Institutions Attended:

University of Science and Technology of China, Hefei, Anhui, P.R. China 1978 to 1983

Degrees Awarded:

B.Eng., University of Science and Technology of China, P.R. China 1983

Publications:

1. Dimopoulos, N.J., Huber, B., Li, K.F., Caughey, D., Escalante, M., Li, D., Burnett, R., and Manning, E., "Modelling Components in DAME," In Proceedings of the Third International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, Vol. II, pp. 716-725, Charleston, South Carolina, July 15-18, 1990.

2. Huber, B., K.F. Li, N.J. Dimopoulos, D. Li, R. Burnett, and E. Manning, "Modelling Signal Behavior in DAME", Proceedings of the 1990 International Symposium on Circuits and Systems, Vol. 2, pp. 1497-1500.

3. Huber, B., Escalante, M., Caughey, D., Dimopoulos, N.J., Li, K.F., Li, D., and Manning, E.G., "Microprocessor Components and Signal Behavior Modelling in DAME," In Proceedings of the 1990 Canadian Conference on Electrical and Computer Engineering, Vol. 1, pp. 27.1.1-27.1.4, Ottawa, Canada, Sep. 4-6, 1990.

4. Dimopoulos, N.J., Huber, B., Li, K.F., Caughey, D., Escalante, M., Li, D., Burnett, R., and Manning, E., "DAME: An Expert Microprocessor-Based-Systems-Designer. An Overview and Status Report," In Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing", Victoria, Canada, May 9-10, 1991.

5. Escalante, M., Dimopoulos, N.J., Huber, B., Li, K.F., Li, D., and Manning, E.G., "Generic Design Rules for the Design of Microprocessor Based Systems in DAME: Bus Arbitration Subsystem," In Proceedings of the 1991 IEEE International Symposium on Circuit and Systems, Singapore, Vol. 5, pp. 3166-3169, June 11-14, 1991.

6. Escalante, M.A., Huber, B., Dimopoulos, N.J., Li, K.F., Li, D., and Manning, E.G.,"Bus Arbitration Modelling and Design in Dame: an Expert Microprocessor-Based-Systems-Designer", To be published in the Proceedings of the International Symposium on Artificial Intelligence, Cancun, Mexico, November 11-14, 1991.

Partial Copyright License

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its user. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

The DAME Editor: A User Interface for Data Acquisition in an Expert Microprocessor-based-Systems Designer

Author [redacted]

(Signature)

DONGNI LI

(Name in Block Letters)

Sep. 20, 1993

(Date)