Order Number 9114052

A generic user interface for hierarchical knowledge-based simulation and control systems

Chow, Alex Chung-Hen, Ph.D.

The University of Arizona, 1990

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

A GENERIC USER INTERFACE FOR HIERARCHICAL KNOWLEDGE-BASED SIMULATION AND CONTROL SYSTEMS

by

Alex Chung-Hen Chow

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

WITH MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1990

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Chung-Hen Chow

entitled A GENERIC USER INTERFACE FOR HIERARICHICAL KNOWLEDGE-BASED SIMULATION AND CONTROL SYSTEM

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Bernard P. Zeigler — Date 11/19/90

Jerzy W. Rozenblit — Date 11/19/90

Francois E. Cellier — Date November 19, 1990

Date

Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

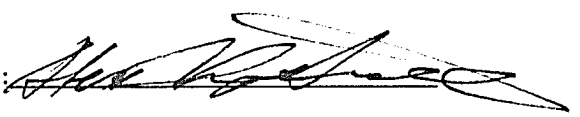
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director Bernard P. Zeigler — Date 11/19/90

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: 

ACKNOWLEDGMENTS

The author wishes to thank Dr. Bernard P. Zeigler for his invaluable advises and consistent support throughout the research. He also appreciates the precious suggestions from Dr. Francois E. Cellier and Dr. Jerzy W. Rozenblit. This research was partially funded by the NASA-Ames under co-operative agreement NCC 2-525 and National Science Foundation under contract CCR-887141148.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Successful design of the user interface for an interactive system must be sensitive to activities supported by the software and the time users spend in these activities. This dissertation presents a systematic approach to user interface design based on this idea. A *control flow model* for the interaction between the human and the application is developed and serves as the framework for the proposed approach. Sub-models of the *control flow model* correspond to actual programming modules and determine the logical system design sequence.

The *System Entity Structure* language is employed to span hierarchical spaces. User interface design is based on the space dimensions of the *System Entity Structure* pruning process. A Virtual Tree Environment is developed on the TI Explorer to provide a programming facility for implementing user-interfaces. Following the proposed methodology, a system designer can give users an efficient user interface to applications that can be hierarchically described by the *System Entity Structure* language. The concept can also be implemented on other window systems such as Microsoft Windows and X-windows.

Index terms: System Entity Structure, Hierarchical Systems, Computer Graphics, User Interface Design.

CHAPTER 1

Introduction

User interface design is an important and difficult problem[BB87]. Literature such as [RH84] [WPSK86] [Nor83] [GL85] offers many design guidelines and methodologies but still leaves many design issues unresolved. User interface design is thus regarded as an art as well as a science. Although iteration of the design approach has been suggested to achieve a better design, this leads to high design cost and inefficient time management that engineers try very hard to avoid.

A valid behavior model of the whole interaction system can provide logical explanations about the difficulties in the design process and why some designs are more successful than others. More important is that the model can be used to predict how the final user interface will behave. The prediction can rectify the design process in the early stage to avoid "bad" designs that are costly to change in a later stage. The predictive method is much better than simply obeying a list of guidelines and design principles. All design methodologies agree on the importance of analyzing the application as well as the user interface itself. We shall develop a control-flow model for human-system interaction that includes both the analysis of the application domain and the effects of the user's activities. The submodels of the control flow model are further mapped to the design of software

modules. The design sequence within each module allows a systematic approach to the user interface design.

Application systems with similar characteristics should lead to similar user interfaces. Hierarchical knowledge-based systems have many characteristics in common. The user interface design of a system should take advantage of this commonality and avoid reinventing the wheel each time. *User Interface Management Systems* are often used to help user interface designs. However, there are only a few *User Interface Management Systems* specifically designed for the implementation of hierarchical knowledge-based systems [HK88] [RT89]. Considering that many modeling, simulation, and control systems are hierarchical, special design tools or environments similar to *User Interface Management Systems* are needed. Exploiting the regularities of hierarchical systems, it may be possible to do an automatic generation of the user interface.

An analysis of a hierarchical application system is based on a knowledge representation language called the *System Entity Structure*(*SES*)[Zei84]. *SES* is a modeling language that describes hierarchically decomposable physical systems such as computer systems, buildings, and robot systems. *SES* also can describe conceptual hierarchical systems such as tree systems and special-purpose languages[KZ89]. By slightly modifying the *strict hierarchy* axiom of *SES*, we obtain a more expressive *Recursive System Entity Structure* for conceptual hierarchical systems. Adapted to the control flow approach, the *Recursive System Entity Structure* defines the underlying hierarchical application space and the system to be implemented.

In *SES*-tree systems, the *dimensions* of the *SES* space are derived and mapped to the input and output primitives of hierarchical systems. The primitives are then mapped to a set of the user-interface operations. Operation hooks of *SES* nodes provide the flexibility to change the mappings and thus tailor the generic user interface to application-oriented user interface styles.

The final user interface is built on a visual language called *Visual-Tree*. *Visual-Tree* offers a conceptual-level interface between the user and the system. Outputs of the *Visual-Tree* on a screen display are graphic objects and are managed as layers or windows. Events in the interactions between the system and the user are described by the operations with the graphic objects on the *Visual-Tree*. Operation mappings translate tree operations into semantic operations. The *Visual-Tree* environment eases the design of a user interface; it encourages the designer to organize graphic objects in a high-level conceptual structure rather than in low-level graphic primitives. The high-level primitives reduce design efforts greatly and still offer the flexibility of having different user interface styles.

Stages in the user's activities are important to the efficiency of a user interface. The mappings of operations from the application space to the *Visual-Tree* environment are designed with the consideration of these stages. Many common conventions of today's user interface designs that behave well in the control flow model are applied to the mapping design.

Chapters 2 and 3 introduce the idea of the control flow, the state space, the space dimensions, and the *System Entity Structure*. Chapter 4 analyzes the *System Entity Structure* and shows how it transforms hierarchical systems into state space dimensions.

These chapters are arranged exactly as the systematic design sequence derived from the control flow model. Chapters 5 and 6 describe the design concepts of the *Visual-Tree* graphic interface environment and how final mappings complete the design of the hierarchical interactive system. Chapter 7 introduces, as examples, the generation of the generic user interfaces for the Smart-Home Control System, the *SES* Editor, and the DEVS modeling environment. Chapter 8 presents the conclusion. Appendix A provides details about the actual *Visual-Tree* implementation on a TI-Explorer Symbolic Workstation.

CHAPTER 2

The Model of Human-machine Interaction

To get a deeper understanding of human-machine interaction, a model is used to abstract a system interaction mechanism. Understanding the model gives predictive information and guidelines for systematic design of an efficient interactive system.

Researchers usually view interfaces from both sides: the user and the task domain. On the user side, the emphasis is to understand the user[GL85] and the gulf between designers and users. On the side of the task domain, the emphasis is on the analysis of the task that the machine performs[ND86]. Both sides are very important to the efficiency of a user interface. Thus a model of the interaction control flow must include not only the interface portion but also the underlying application system and the user's activity.

In [BKP90], an approach is proposed to model the human-machine interaction based on the GOMS[CMN83] user model. Two different models are required, a *production system* to describe the user's knowledge of how to use the device or system, and a *generalized transition network* model to represent the system itself. The model can predict empirical results about text editing, learning, performance, and skill transfer but lacks design methodology for the system. We need a more design-related model which outlines a systematic design approach as well as gives the predictability of design results.

Before creating the interaction model, we have to decide on the right level of abstraction for our objectives. For example, in user-machine interaction, we can choose levels of abstraction from several different perspectives, such as processing time, functionality, implementation, etc. The objectives of our approach relate to the design of efficient user interfaces. The efficiency of a user interface is measured by the time spent to perform particular tasks via the user interface. To model the time spent in various activities, we follow the control flow of the system. The total time spent is the sum of the times consumed at each stage in the control flow. Similar to the design phases mentioned in [FvD82] [NS79], the control flow can be broken into the following stages: the state space, the mapping model, the interface channels and the human activities. Each stage consumes a portion of the interaction time. The state space, the mapping model, and the interface channels are on the side of the task domain and the human activities are on the user side.

2.1 State space model of task domain

The kernel portion of the control flow is the task domain. This part of the flow is modeled by using the state space concept. The concept is an extension of the state transition network model[KP83]. A state space is the set of all possible system states. A state can only change to another state via state transition vectors; an initial state and a set of state transition vectors define the state space that can be either finite or infinite. Often, the path a state transition vector takes can be regarded as a dimension of space. The transition made through transition vectors moves the state along the corresponding dimension.

Any application can be represented by a state space. An application task is represented by a state point in the space. To complete the task is to transit from an initial state to the final state representing the task. The transitions can be either autonomous or controlled from the outside world. Yet, it is often difficult or even impossible to design a system that can transit to a goal state completely autonomously due to the nature of the application. For example, in a VLSI layout system, it is difficult for software to generate optimal layouts for special purpose VLSI chips. Usually, we leave such problems to the user and let him/her make the decisions of how to transit from the initial state to the goal state. The interactions between the external world and the state space are the controllers of the state transitions from the external world. When the transition is controlled by human users, the system is called user interactive.

The state space model defines the tasks of an application. A well-modeled task domain leads to a successful implementation of the final system. For example, the VISICAL spread sheet package doesn't have an excellent user interface but the implementation is very successful[ND86] due to its good task model. However, it is not easy to model the task domain completely and correctly. The system designer of a particular application must have sufficient understanding of the task domain to create a proper task model.

To define the state space of a given application domain, the designer has to find the *range* of the space first. The range can be found in two ways. One is to find all the possible task states of the application; the other is to find the proper state dimension ranges. The first aspect is usually more direct than the second one in applications with a few task states. For complex applications, the second method is always more systematic

than the first one. From the defined ranges, the state transition vectors can be found. Once the corresponding dimensions and their ranges are identified, the unit transition vector on each dimension can be easily defined.

Besides the state transition vectors, a designer also must assign a suitable value to the initial state; this initial value is the origin of the space. The state space is defined by the space origin and the state transition vectors as shown in Figure 2.1.

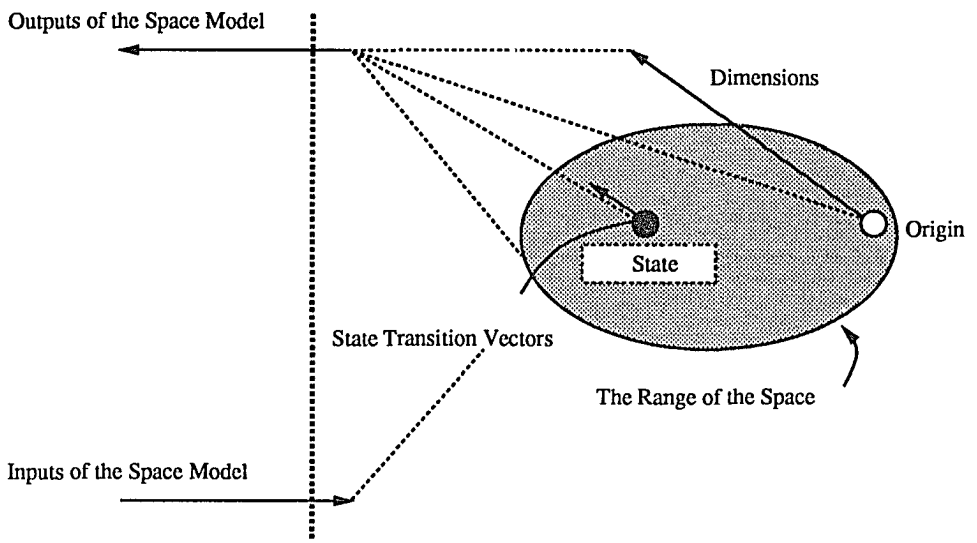


Figure 2.1: Space model

For example, controlling a light is an application domain. Tasks of the domain are turning on and off the light. The state of the light is either on or off. To perform the task of turning on or off the light, the state must be transited either from on to off or from off to on by the state transition vectors on-to-off and off-to-on, respectively. If the space is to be defined by dimensions, there is only one dimension needed with the range of on and off.

The inputs and outputs of a model are used to interface with other models. Because the space model is interactive in the control flow, it has both inputs and outputs. Clearly, the inputs of the state space control the state via the state transition vectors. For a non-dimensional state space, one unique input controls a transition vector and enables the outside world to access the state space. For a dimensional state space, each unit vector of a dimension has an input that guarantees complete accessibility of the state space.

The outputs of the space give the outside world the position of the current state and the shape of the state space. Each dimension has the following information to be output: its direction, range, and the value. The position of the current state is the composite information from the dimensions and their values. The ranges and the values give the position of the state. The ranges and the directions outline the shape of the state space and define the available state transitions.

2.2 Model of the interface channels

Different physical forms between the task space and the user always introduce gaps in the interaction. Interface channels are provided to bridge the gaps. In the user's world, the physical forms are signals perceivable by a human such as video images and audio sounds and movements of a physical object. In the task space, the forms of the world of computers are the electronic signals such as voltage and current. The interface channels provide two-way translations between the two and are logically divided into two groups: the input and the output channels. The input channels are keyboards, mice, joysticks,

etc., that translate physical-world signals to electronic signals. The output channels are the video screens and the audio speakers that translate in the reverse direction.

There are many limitations of the interface channels. Software manipulations are required on top of these channels to eliminate the need for hardware management by software designers. The software manipulations are called channel buffering. Channel buffering handles things like translating keystroke signals into software codes or converting codes into images on a screen. Channel buffering includes the software that provides higher abstractions of the interface channels. For example, the parsing of a command line or the display of a graphic icon is a higher abstraction of the interface channels.

Most channel buffering software, such as X-windows and Macapp[Sch86], provides higher-level channel primitives. Because the buffering software tends to allow all the possibilities of user-interface styles, system designers often find that the level provided is still too low. Having a channel buffering software that provides both flexibility and an easy-design level is necessary for fast prototyping. A higher designing level also provides additional dimensions on the interface channels such as the haptic channels described in Chapter 8 of [BB87]. The dimensions, such as windows, cursors and icons, are created by the channel buffering software.

As Figure 2.2 shows, there are two aspects of inputs and outputs for the model. One is with the physical world, in which the outputs are the video and audio signals and the inputs are the keyboard, mouse, etc. The other aspect is the digital world, in which the outputs are the channel-buffered input data to other software modules and whose inputs control the physical output signals.

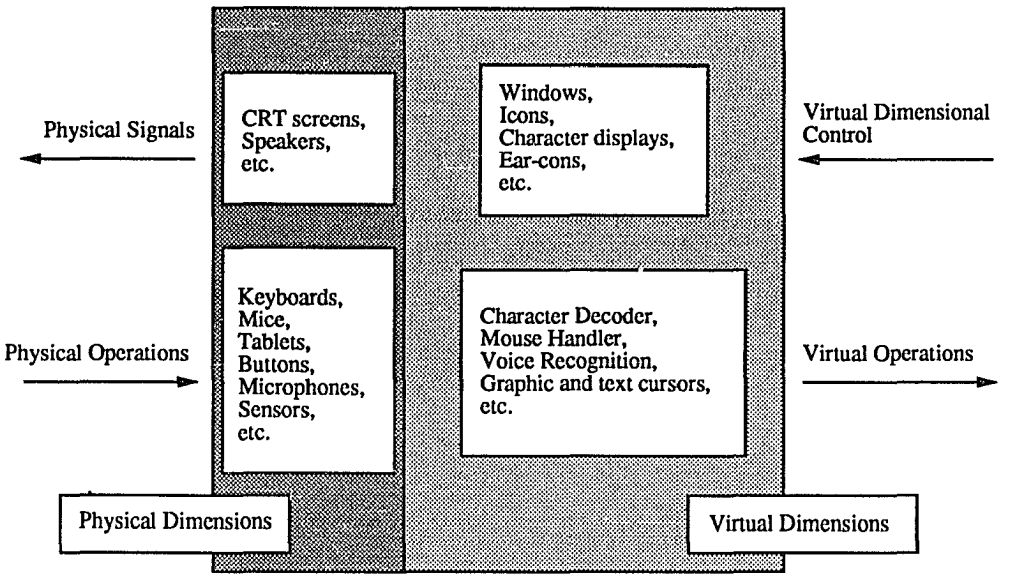


Figure 2.2: Interface channel model

2.3 Model of mappings

There are three topics in this mapping model based on different considerations: level mappings, dimension mappings, and multiple mappings.

The level of space transition vectors is usually too low for the human user to control directly, and the output of the space is overly abstract for the user to understand. The signals from both the interface channels and the task space must be mapped to proper semantic levels so that the space and the user can interact with each other at their separate, comfortable levels. Thus, the input mapping module maps the high-level semantic commands into low-level inputs of the state space and the output mapping module maps the outputs of the space state to outputs meaningful to humans on the output channels.

Mostly, the available dimensions of the interface channels are much fewer than those of the state space and their appearances are different. The mappings must always ignore certain application dimensions and put only the most significant dimensions onto the interface channels. The selection of the significant dimensions and their mapping onto the other set of dimensions in different appearances are important topics in the interactive system design.

As the last consideration of the mappings, an operation can appear in many different forms. For example, we can either control the same operation from a keyboard, from a mouse or even from a microphone. A state can be shown on the output channels in different appearances, such as the video and audio beep for error messages. These kinds of mappings are called multiple mappings.

These mappings are related to "user interface style" and are responsible for the presentations of the space on the output channels and the style in controlling the state transition vectors via high-level semantic operations. The mapping model is a state space in itself but it is not application-related. The mapping state and space only relate to the behavior of the mapping model in things like the mapping styles and the selected mapping dimensions; some of them are under the control of the user and share the same interface channels with the application state space. The user controls the behavior of the mapping model by changing its states. The ranges and dimensions of this mapping space are limited and of high abstractions compared to those of the application space. The mappings of the *mappings' space* itself onto the interface channels are much simpler and more direct than those of the application space. Obviously, the designer must separate

the two different state spaces by mapping them onto different dimensions or ranges on the interface channels.

The design of the mappings decides the efficiency of the user interface because the time spent in user activities depends heavily on the styles of the mappings. Design considerations based on user activities are very important for this model.

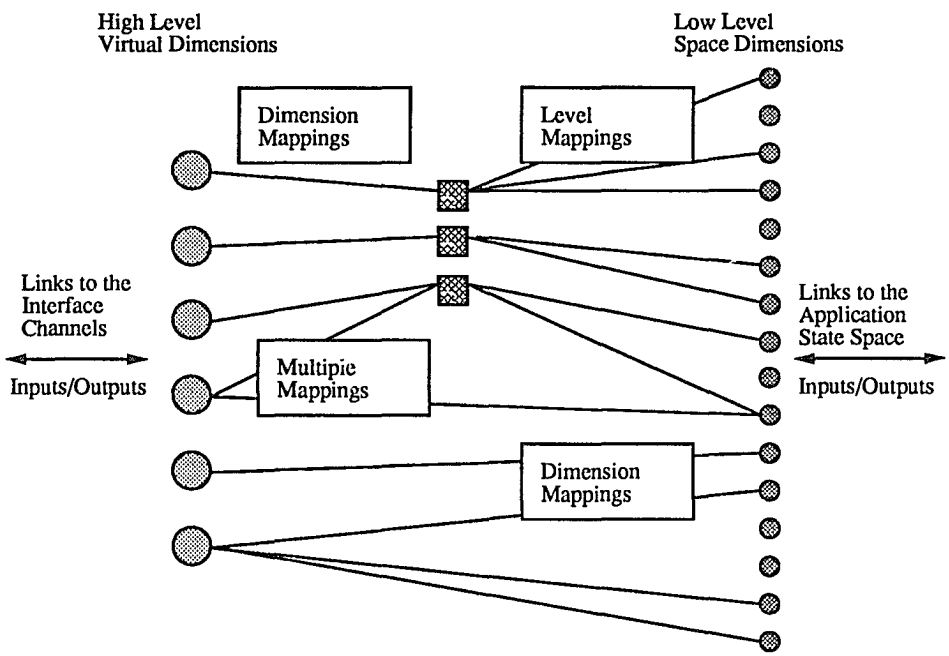


Figure 2.3: Mapping model

As Figure 2.3 shows, this model has the similar inputs and outputs as the interface channel model except that its two sides are now the different levels and different interface styles.

2.4 Model of user activities

Previous models describe the interactive system which is only one side of the interface. The other side of the interface is the user. The activities of the user side are modeled as the human processor in [Car84] or as the stages of user activities in [ND86]. The stage model is more control flow oriented and the activities (in Figure 2.4) are:

- Establishing the goal
- Forming the intention
- Specifying the action sequence
- Executing the action
- Perceiving the system state
- Interpreting the state
- Evaluating the system state with respect to the goals and intentions

The primary stage is the establishment of the goal. The goal point exists both in the conceptual mind of the user and the task space. To carry out an action requires three stages: forming the intention, specifying the action sequence, and executing the action. To assess the effect of the action also requires three stages, each in some sense complementary to the three stages of carrying out the action: perceiving the system state, interpreting the state, and evaluating the interpreted current state with respect to the original goal and intention state in the space. The whole process is to bring the current state to the goal state by executing the action, that is, controlling the transition vectors.

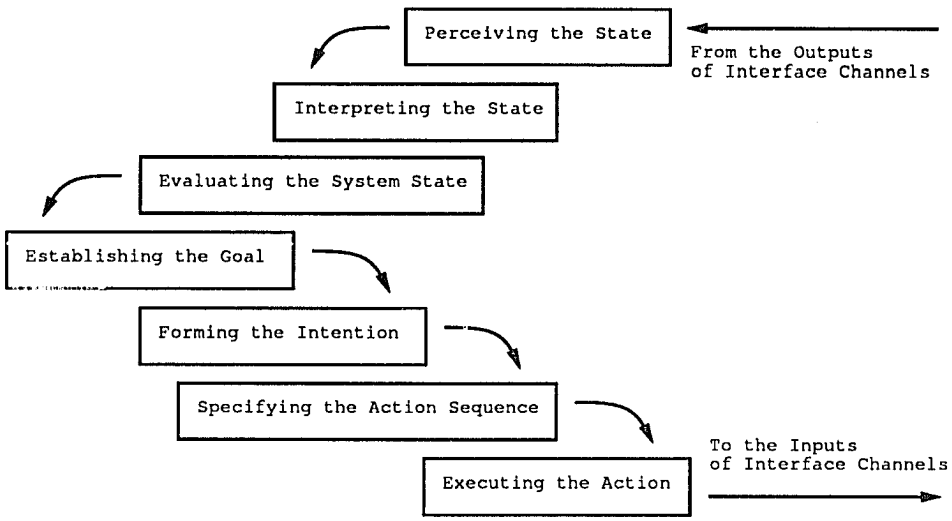


Figure 2.4: The model of the stages of user activities

The evaluation of the user-friendliness of a user interface occurs during these user activity stages. The activities occur in the mind of the user. Usually, the time spent in passing controls through previous system models is not significant because of the fast machines currently available, though. Time spent should still be limited within 0.2 second[SM84]. However, the time spent on user activities depends largely on the design of these system models. Many principles of user interface design become apparent when designers take user activities into consideration.

2.5 The non-autonomous interactive system

The user-machine interaction mechanism includes all the models described. They are connected as Figure 2.5 shows. There are two models in both sides of the interface

connected by the inputs and outputs of each model. The state space outputs the space and state and receive the controls of the state transition vectors via the interface channels.

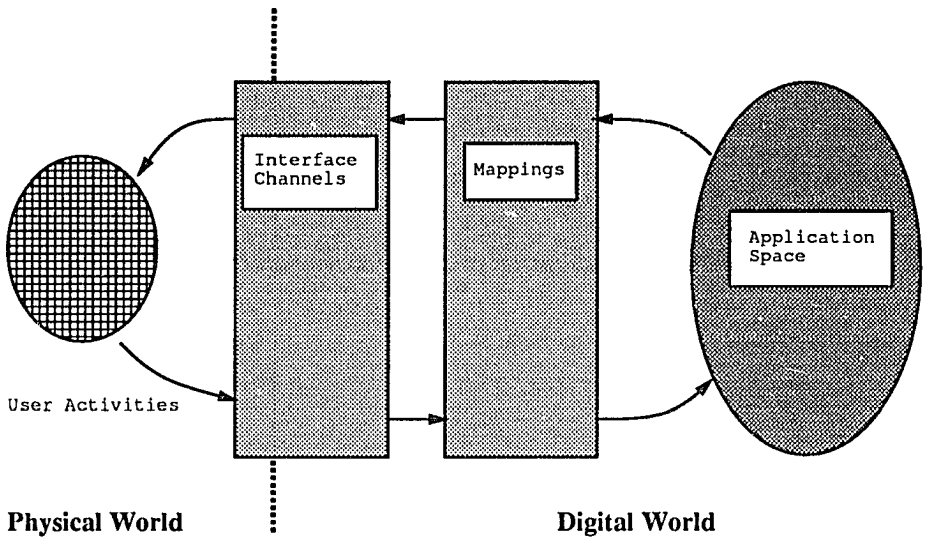


Figure 2.5: The entire picture of the interaction model

The interaction is initiated from the user's intention of a goal state. The user executes the action to the state space to bring the state closer to the goal. The space has the flow control until its state transition is made. The outputs of the space are mapped to the output channels where output channels have the control and translate the outputs for the user. The user has the control after the perception of the outputs and then starts the interaction again. The entire control flow of the interaction mechanism forms a loop that repeats until a goal state in the user's mind is reached. Note that the application space described here is a non-autonomous one; every state transition is totally controlled by the user.

The application space has all the semantic meanings of a task domain. The state in the space represents the task in the application. A completed task is usually recorded as a state of the space. To keep the state as a reusable result, the designer must offer the user methods to save and retrieve a partial or entire state.

2.6 The other dimensions: autonomous behaviors

In an autonomous system, there is no need for user-machine interaction. Recall that autonomous systems can always reach goal states. Autonomous systems are, of course, preferred by human users because applications can be automatically completed. However, there will always be non-autonomous systems because of the complexity of applications.

Some semi-autonomous systems help the users by transiting automatically to states that are closer to the goals; but they still cannot reach the goal states autonomously. For example, the router of a VLSI layout system may help the user get part of the routings in the layout designed. The time varying and non-user controlled behavior of the autonomy is modeled as a small feedback loop from the output to the input of the space model through an auto-transition model. This model receives the outputs of the space, applies the transition algorithms, and decides what states to transit to as Figure 2.6 shows.

An auto-transition model controls the flow from the user to the input channels. When the state of the space should be changed autonomously, this part of the system will take over control unless the user intervenes during this autonomous period. The state of the space is stable when there are no more transition algorithms applicable and the controls from the user to the input channels is again connected.

Semi-autonomous systems have the intelligence to help the user reach a goal state faster, but autonomous actions occur only when the goal state is definable and path finding algorithms to the goal state exist.

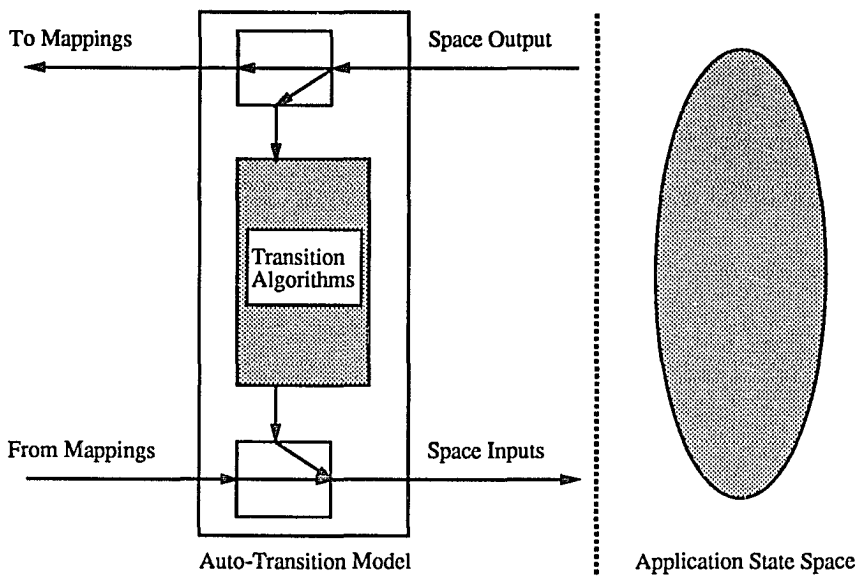


Figure 2.6: The semi-autonomous interactive system model

Similar to the mapping state space, the auto-transition model has its own state space. The user also controls the state of this space which is usually simpler than the application space itself. The inputs and outputs of this space are also mapped to certain dimensions on the interface channels.

2.7 Design based on Control Flow Model

The control flow model is an abstraction of the time spent during each control flow stage. For example, the human activity model abstracts the time spent by the user. Each

submodel is a direct representation of the execution time of a software module or the response time of a physical object. This model also represents the control structure of a software system. A logical software-design sequence can be derived from this model.

One purpose of this model is that the designer can learn more about the interaction mechanism and predict the result for a particular design decision. It is assumed that a good user interface is always task efficient. The efficiency is measured by the time spent in the interaction loop for a task. That is the time for a user to move the initial state to the goal state. The less time the process spends the better the user interface is. Of course, the efficiency depends much on variables such as the difference between users and the difference between task domains. There is no analytical way to find an interactive system design with optimal efficiency. It is the designer's responsibility to define the variations and find the proper compromises. That is why the specification of the intended task domain and the identification of the intended user group are important before doing system design. It is still important to design a system that is efficient for all levels of users. The sensitivity of a user interface efficiency to these deviations is another important topic.

Even if there are no variation, it is still impossible to get an optimal design. The heuristic approach we propose is to evaluate each design alternative by the time spent within the control-flow model and use the alternative that requires the least overhead and is the least sensitive to the environmental deviations, because this control flow model abstracts nicely how each component model incurs its time overheads. Why certain user interface styles are more efficient than others also can be easily explained under the time cost concept of this model. If we can come up with an alternative that leads to a less

total cost, especially the time spent during user activities, we are on the right track to an efficient user interface.

In this model, the design sequence of an interactive system and its user interface is given in several stages, with each stage corresponding to the realization of a submodel in the control flow. Alternatives related to the design of each stage should be considered under the time overhead concept of the control flow model. The first stage is the modeling of the application domain as a state space, that has all the semantic representations of the application. Without this stage, the system cannot even perform a proper task for the application. The second stage is the modeling of the interface channels and the channel bufferings. The important thing here is to find all interface dimensions available and expand them to provide the needed flexibility and expressive power. The final stage is the design of the mapping models. This part of the design is responsible for transferring the low level space inputs and outputs to the high level semantic operations and presentations on the interface channels for human users. This is the most vital design stage for the user interface style. User activities are the major contributors to user interface efficiency. Decisions made in each realization stage are evaluated by the effects on the activities.

As an example, we design a user interface to control a light. We can see the systematic design sequence as Figure 2.7 shows. First, find the state space, on and off, of the application domain and then study interface channels that provide the user the accessibility to the state space. Even with such a simple example, we encounter several design selections. For example, the input channel dimensions mapped by the transition vectors can be a command line language that offers, say, "turn on" and "turn off" commands. To learn

to use the command line language increases the time spent when the user specifies the intended actions. The simple click of a mouse is faster than a command line language in such a case. Thus, the choice of this mapping is made to be the clicking switches on the screen. We also have several choices for the mapping of the state to the output channels; we can display the state in text mode or in graphic mode. If the display is in text mode, the user has to take a longer time to interpret the state expressed as either "off" or "on" than simply perceive the white- or black-colored image on the screen. Thus, we select the graphic display of a white or black oval to be mapped as the on or off state, respectively. Similar decisions must be made throughout each design stage based on the time spent in the control flow model.

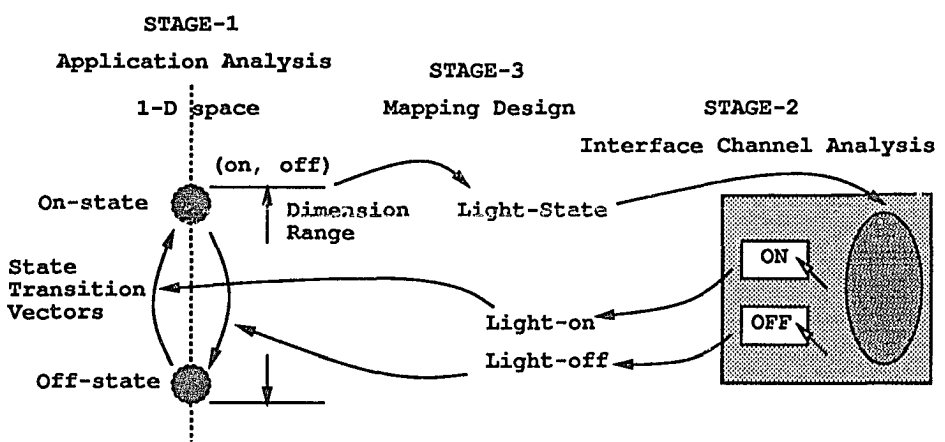


Figure 2.7: The control-flow design of the light system

The following chapters discuss the user interface design of hierarchical systems by using the control-flow approach. More complete examples are given.

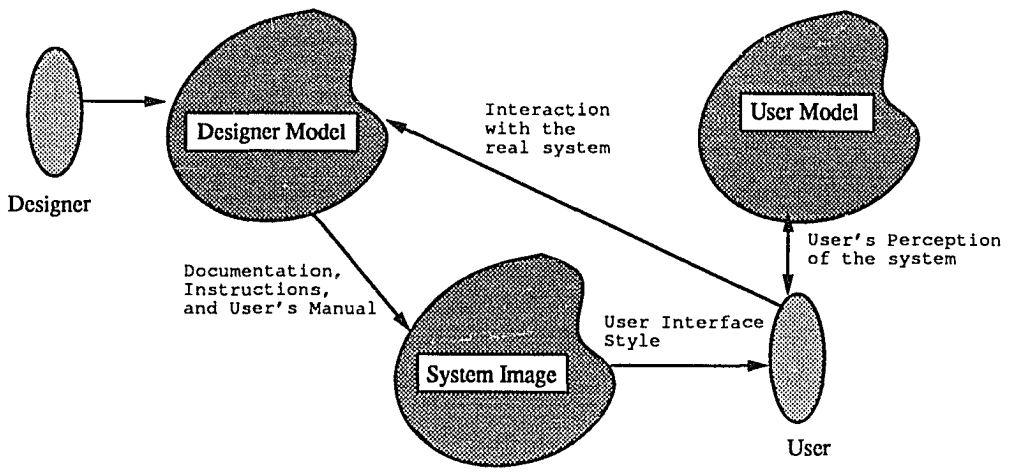


Figure 2.8: The real system and the three models

In implementing and using interactive systems, there are three different models according to [ND86] due to the nature of the interaction: the Designer Model, the User Model, and the System Image (Figure 2.8). The Designer Model is the conceptualization of the system held by the designer and the conceptual model of the system to be built. The design approach and the final system are the Designer Model. The System Image is the image resulting from the physical structure that has been built, the documentation, and the instruction. The user develops a mental model (User Model) of the system from the way he/she interprets the System Image. The Designer Model fits the system actually implemented best. For the interaction to be efficient and useful, the three models should be kept as closely as possible to the real system. These considerations can be explained via the control flow model. The knowledge of the correct real system implemented in the user model reduces the time spent in interpreting the perception and the specifying of actions in the user process model.

CHAPTER 3

System Entity Structure

Before one can design an efficient interactive system, the application domain of that system must be studied completely to obtain a proper model. Here, hierarchical simulation and control are the application domain. Application domains of combined continuous/discrete simulations have been analyzed in [Cel86] [CWZ90]. Zeigler proposed the *System Entity Structure*(*SES*) as a language for hierarchical-system modeling, simulation, and knowledge representation in [Zei90] [Zei84]. The *SES* language is an excellent tool for the analysis and description of a hierarchical-system domain. This chapter introduces the *SES* language.

The representation scheme of the *SES* language supports the following three relationships: *decomposition*, *taxonomy*, and *coupling*.

Decomposition: describes the manner in which an object is decomposed into components. The hierarchical character of the *SES* is that components can themselves be decomposed into subcomponents, and so on, to a depth determined by the designer's objectives.

Taxonomy: organizes the different kinds of objects, i.e., how they can be categorized and subclassified.

Coupling: represents how models are coupled and what constraints apply to component combinations.

3.1 Definitions and Axioms

The *SES* is a labeled tree with attached variable types that satisfies the following axioms:

Uniformity: Any two nodes that have the same label have identical attached variable types and isomorphic subtrees.

Strict hierarchy: No label appears more than once down any path of the tree (modified in the recursive *SES*).

Alternating mode: Each node has a mode that is either entity, aspect or specialization; if the mode of a node is entity then the modes of its successors are aspect or specialization. It the mode of a node is aspect or specialization, then the modes of the children are entity. The mode of the root is entity.

Valid brothers: No two brothers have the same label.

Attached variables: No two variable types attached to the same item have the same name.

Inheritance: every entity in a specialization inherits all the variables, aspects and specializations from the parent of the specialization.

An entity usually represents a real world object that can either be independently identified or postulated as a component of a decomposition of another real world object. An aspect represents one decomposition out of many possibilities of an entity. The children

of an aspect are entities representing components in a decomposition of its parent. A specialization is a mode of classifying entities and expresses choices for the components in the modeled system. The children of a specialization are entities representing variants of its parent. It is not unusual that many entities of the same kind exist within a system; the multiple entity is a way to represent such situations concisely. The example illustrated in Figure 3.1 is the *SES* of a simple computer architecture.

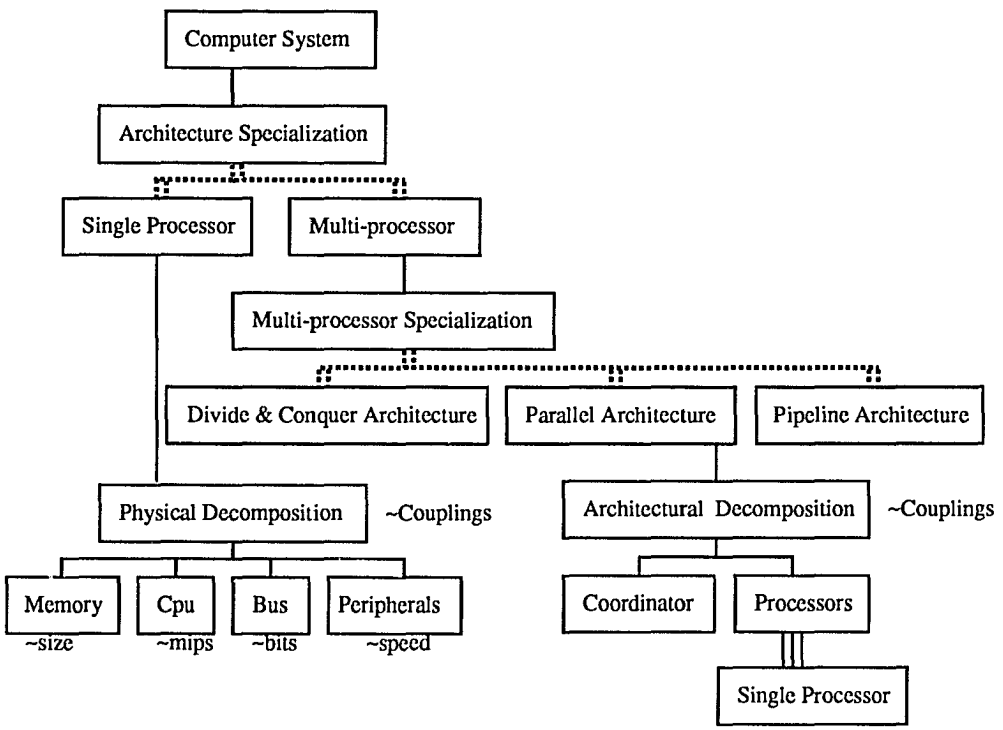


Figure 3.1: The *SES* of a simple computer architecture

The computer system can be specialized into either a single-processor architecture or a multiple-processor architecture. For any single-processor architecture, the computer is

decomposed into memory, cpu, bus, and the peripherals. Each entity has its own attached variable type. For example, the memory has size as its attached variable type. The multiple-processor architecture can be further specialized into pipeline architecture, divide-and-conquer architecture, and parallel architecture. Under the parallel architecture, a computer system can be decomposed into coordinator and many single processors.

3.2 Pruning process of an *SES* tree

A pure entity structure is one having no specializations and at most one aspect hanging from each entity. The pruning process is used to create such a pure *SES*. The intermediate result of a pruning process is a *Pruned Entity Structure*(*PES*) that contains fewer aspects and specializations than the original and therefore specifies a smaller family of alternative models than the latter. Ultimately, the pruning process terminates in a pure entity structure that specifies the synthesis of a particular hierarchical model.

To be more specific, the physical pruning process is described below. In visiting each entity nodes of the *SES* tree under pruning, three terms are used:

Specialization group: the set of specialization nodes of the visited entity node.

Aspect group: the set of available aspect nodes of the visited entity node.

Variable type group: the set of attached variable types of the visited entity node.

The original elements of the above groups are the respective elements under the entity node when the pruning process begins. At each entity node, there are four types of processes involved until a pure *SES* tree can be obtained:

Specialization:

- Select one specialization node among the specialization group and delete the selected one from the group.
- Select an entity node from the children of the selected specialization node. According to the inheritance axiom, all the specialization nodes under the selected entity node are now added to the specialization group of the original entity node. The label of the original entity is prepended by the label of the selected entity and an underscore. For example, if the "computer-system" in Figure 3.1 is specialized to "single-processor" then its label is changed to "single-processor_computer-system". Both the aspect group and the variable type group have similar inheritance from this process.
- Repeat the above two steps until the proper specialization sequence is made or until there is no element in the specialization group. The proper specialization gives the desired aspect group and the variable type group.

The main purpose of the specialization process is to reach a proper place for the desired decomposition through inheritance. After the specialization, there is either no more specialization node or the specialization group is not important for the application.

Decomposition: select an aspect node from the aspect group after specialization. Delete all others so that the selected one is the only aspect node under the visited node. It decides what decomposition of the visited node will be.

Setting Attached Variable: The attached variables are assigned values. Each variable in the variable type group is optionally given a proper value.

Cloning: If the entity node visited is a multiple entity, it can be cloned into one or more instances under the parent of the original multiple entity. If the cloning process is made after specialization, decomposition, or variable setting processes, the effect of the processes on the original entity will be the same on the cloned one. That is, the cloned one has all the same specializations, decompositions and attribute settings as the original one. The number of copies of the multiple entity can be zero or greater. The creation of a "clone" is needed if a pruning process is required on this node.

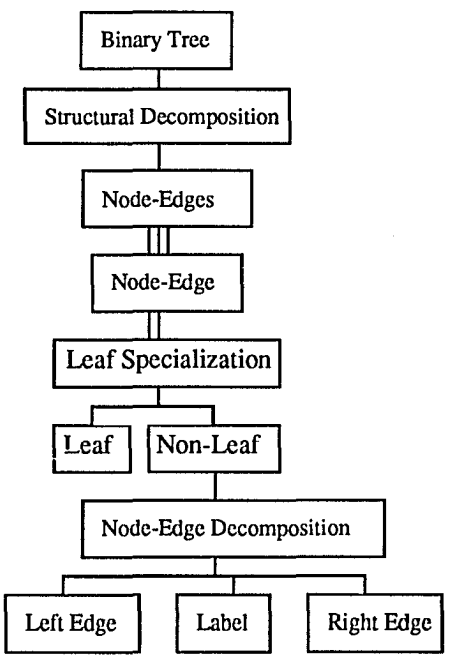


Figure 3.2: The *SES* of the binary trees

The iteration of visit and pruning makes the family of alternative models smaller and smaller until the *PES* is finally a pure *SES*. Currently, the user-interface to the *SES* and its pruning process is implemented as ESP-scheme [Zei87] [RHKZ90] which is basically a programming-language user-interface.

As Figure 3.2 shows, we can use the *SES* to describe the class of binary trees. Any binary tree can be decomposed into many *node-edge* entities. A *node-edge* entity has a name and two edges, left and right. Each edge is a pointer or label to another entity. The *SES* shown describes the composition of the binary tree. The hierarchical aspect of the binary tree is indirectly expressed by the edge pointers. We can see that the *SES* also describes the implementation aspect of the data structure of a binary tree system. The number of possible structures of a binary tree is infinite. The multiple entities express the infinite possibilities of the tree structures even though the *SES* is a finite tree.

3.3 Recursive *SES*

The *SES* can completely describe and model physical world objects, but the finite tree characteristic limits its ability when it comes to the conceptual world. A decomposition path of a physical object is always finite and is limited by the objectives of the modeler. The aspect nodes represent the hierarchical relationship of the decomposition path directly. It is possible that a decomposition path of a conceptual object is infinite.

For example, the sentences from a recursive grammar can have an infinite decomposition path. We cannot use the aspect nodes to describe the recursive hierarchy because of

the axiom of strict hierarchy. Nodes with the same label cannot occur in the same decomposition path more than once. We can get around this problem by using multiple entities to express the infinite recursion just like in the previous examples, but the expressiveness of using the aspect nodes for hierarchical relationships or sequential relationships will then be lost. The *SES*'s description no longer conveys the hierarchical meanings directly by the aspect nodes. Instead, the hierarchy relationship is expressed by the entity nodes.

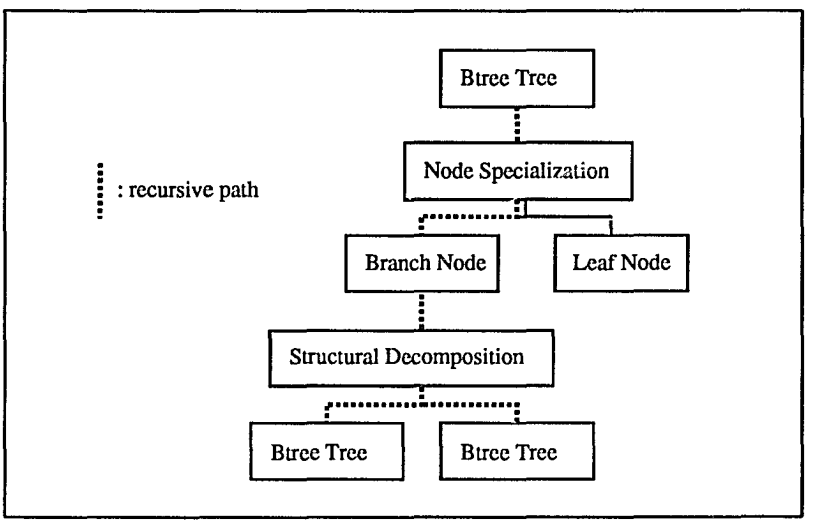


Figure 3.3: The recursive *SES* of the class of binary trees

The binary tree structure in the recursive *SES* is shown in Figure 3.3, as an example, in contrast to the *SES* without recursion. Two nodes with the same label occur in the same path that describes the recursion. Another difference is that the aspect nodes now describe the hierarchical relationships. The new *SES* displays not only the composition but also the hierarchical relationships of a tree system. The recursion ends by a selection

of the specialization path with non-recursive leaf nodes. Thus, the hierarchical axiom is now relaxed to be:

Hierarchy axiom: A label can appear at most twice down any path in the hierarchy. The path with reappearing labels is called a *recursive path*. There must be at least one multiple entity on this recursive path or a non-recursive specialization path to terminate the recursive path.

The modification gives the *SES* a recursive view of the world. It also gives the *SES* the expressiveness to describe an infinite conceptual system hierarchically.

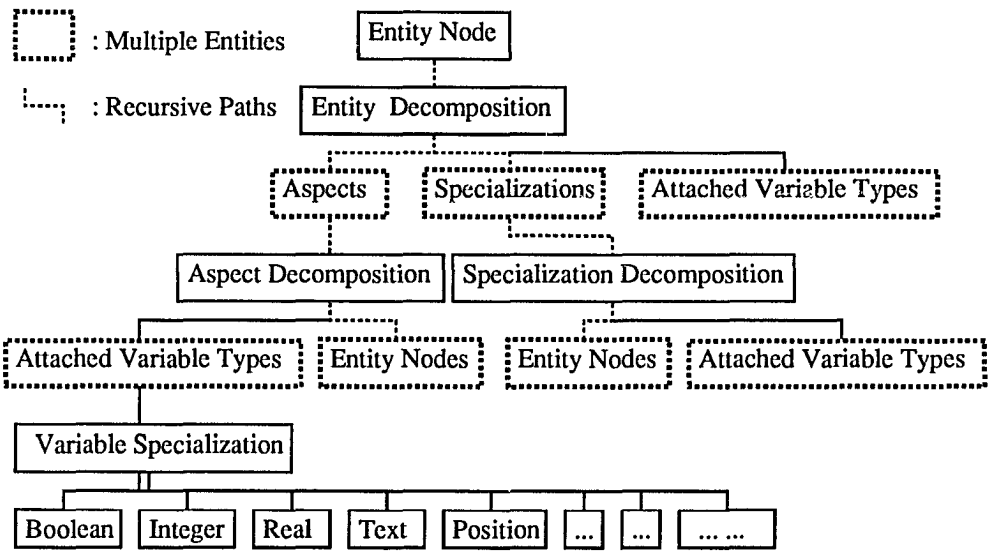


Figure 3.4: A *SES* of the *SES* using *Recursive System Entity Structure*

3.4 *Recursive System Entity Structure* of *SES*

As an example, we give a *Recursive System Entity Structure* description of the *System Entity Structure* itself in Figure 3.4. The root of the tree is an entity node that has aspect nodes, specialization nodes and its attached variable types as its children. The aspect and specialization nodes are multiple entities that can have clones of any number of copies. Similarly, the entity nodes are the children of the aspect and specialization nodes. The recursive path starts from the root-entity node down to its aspect and specialization nodes and back to the entity node itself. Note in this case that a recursive path forks a new path each time a new copy of the multiple entity is cloned and ends at a multiple entity without clones.

This *Recursive System Entity Structure* description will again be used in Chapter 7 to create a user interface for the *SES* Editor.

CHAPTER 4

The State Space of Hierarchical Systems

As introduced in the last chapter, the *System Entity Structure* language system can describe classes of hierarchical systems. If an application can be described by an *SES* tree, the *SES* tree describes the application state space hierarchically. For example, if an *SES* tree describes the structure of a file system, the states of the application space are all the possible structures of the file system.

How to define a hierarchical space by using the *System Entity Structure* language is discussed in this chapter. The space defined with the *System Entity Structure* language is called *SES-Space*. The *SES-Space* is analyzed by the *System Entity Structure* definition itself. Once the *SES-Space* is analyzed, we implement the space directly from its dimensions. For each dimension, the implementation includes its value, transition vector and output function. In the final section, a generic implementation is described. The generic implementation supports all hierarchical application spaces that can be described by the *System Entity Structure* language. In fact, the final implementation reads in a *System Entity Structure* description and generates the corresponding *SES-Space* and its input/output primitives. The input/output primitives are important because they allow other programming modules to access the state of the *SES-Space*.

4.1 Analysis of System Entity Structure Space

An *SES* describes a class of hierarchical systems. Each possible system structure is a state. All possible states corresponding to the *SES* form the space called *SES-Space*. Because the states of *SES-Space* are not easy to enumerate, it is easier to analyze the *SES-Space* via dimensions.

An object-oriented concept is applied to manage the state space dimensions; it encapsulates the related dimensions and the related transition vectors. There are two advantages in adopting this idea:

- It provides a systematic and hierarchical management style for complicated space dimensions.
- It cuts down the time spent when the user is specifying and executing the actions. If a task needs the application of several state transition vectors, the user has to issue actions transformed into those low-level transition vectors. By grouping dimensions, the user can execute a single action on a group of closely related dimensions at once instead of repeatedly executing similar actions on several dimensions.

So, the object-oriented concept can make an interactive system more efficient by the above reasoning. I apply this concept in analyzing the *SES-Space* and the designs in other stages. Both the designer and the user have the choice of the related dimensions to be encapsulated. The flexibility to let the users decide the dimensions to be encapsulated enables one to fine tune the interactive system for the highest efficiency. The actions

that a user uses to encapsulate dimensions is another group of dimensions relating to the mappings.

The *System Entity Structure* is itself object-oriented. The object-oriented aspect arises from the tree node structure. A node represents an object to be decomposed into other sub-objects. The decomposition can be performed down to any level via the aspect nodes. All semantically-related attributes at the current level of decomposition are grouped as aspect nodes, specialization nodes and the attached variable types. This object-oriented characteristic of the *SES-Space* dimensions is always kept in the analysis.

4.2 The dimensions of *SES-Space*

The possible structures of a system are bounded once a fixed *SES* tree is given; this means that the *SES* tree defines the *SES-Space*. The state of the *SES-Space* is a pure *SES* tree that represents a single system structure in the structure class. The system structure is obtained by applying the pruning process. Clearly, the dimensions of the state transitions correspond to the behavior of the pruning process. The state transition dimensions are grouped into objects called *pruning nodes*. Each pruning node corresponds to an entity node in the *SES* tree. The entity node is called a definition node because it defines the dimensions and ranges available for the pruning node. At first, there is only one pruning node of the pruning process that corresponds to the root entity node. Each pruning node has the following group of dimensions that are essentially derived from the behavior of a pruning process described in the previous chapter:

Specialization dimension:

The pruning node can be specialized along this dimension defined by its definition node. This dimension is defined from the possible specialization of the definition node.

Decomposition dimension:

This group of dimensions is to decide the kind of decomposition of the pruning node. Again, the aspect nodes under the definition node define the possible decompositions.

Variable type values:

This group of dimensions is the variation of the values of all inherited variable types. The values are defined by the types of the variables.

Structural dimension:

This dimension represents the existence of the pruning nodes and the hierarchical relationships among the nodes.

The decomposition pruning nodes:

New pruning nodes are created according to the entity children of the aspect node chosen. The new nodes are the children of the current node and each has a corresponding entity node as its definition node.

The multiple pruning node:

If the definition node of a pruning node is a multiple entity, the pruning node is called *multiple pruning node*. The number of clones of a multiple pruning node can be from zero to any finite number. Each clone stands for a distinctive

pruning node with the same definition node and, thus, another group of the same dimensions.

To give a more formalized description of the *SES-Space* of a *SES* tree:

$$SES\ Space = SPEC \times DECP \times AVV \times PN \tag{4.1}$$

SPEC : the specialization dimensions

DECP : the decomposition dimensions

AVV : attached variable value dimensions

PN : the pruning nodes created from the decomposition and the cloning

4.3 The states of the *SES-Space*

A state can be represented by the values of the space dimensions. So, the state in the *SES-Space* can be formulated as:

$$SES\ State = (\ spec,\quad decp,\quad avv\ )[pn] \tag{4.2}$$

spec, *decp*, *avv* and *pn* denote typical values of *SPEC*, *DECP*, *AVV* and *PN*, respectively. Since the values of *spec*, *decp*, and *avv* are meaningful only when associated with the corresponding *pn*, the value of *pn* is formulated as a kind of array index to the other three dimension values. Each dimension has its own value type and range as described below:

Specialization State:

The dimension value is the specialization path via its definition node. The specialization path is very much like a stack of layers. Each layer is a pair of specialization

and entity node. The entity node is one child of the specialization node. The range of the specialization node is the specialization group. The original specialization group is the set of the specialization nodes under the definition node before any specialization begins. When a layer of specialization pairs is pushed onto the stack, the specialization group is changed according to the inheritance axiom. So, the range of specialization dimensions changes after each specialization. A layer pushed onto the stack means another level of specialization. It is important to keep the sequence of the layers for the semantic meanings of a specialization process. The stack keeps not only information of each specialization but also its sequences. The value of this dimension is simply the current layered stack.

Decomposition State:

The state of the decomposition is the aspect node selected from its range, the aspect group of the pruning node. The aspect group is similar to the specialization group. Originally, it is the set of the aspect nodes under the definition node. Because of the inheritance, it changes each time the specialization state changes. The transition is better made after the transition of the specialization state to ensure the maximum range available. The value of this dimension is either a selected aspect node from the aspect group or simply *not selected*.

Variable State:

The variable type defines the variable range and value. The set of variable types is the variable type group after the inheritances of all the specialization processes. Each variable in the group associates with a value of its own.

Pruning Node State:

The first three dimensions exist only when a pruning node is created. There are two places that a pruning node can be created. One is in the decomposition state transition. Each time this state transition(selection) is made, new pruning nodes come to exist corresponding to the child entity nodes under the selected aspect node. The other situation is when a pruning node is cloned. One extra pruning node is created corresponding to the same definition node. If there is already a cloned pruning node as a template, all states on its dimensions are copied. Otherwise, the origins of the dimension groups from the definition node are copied as the initial state. The number of clones can be from zero to any finite number. The existence of pruning nodes and their hierarchical relationships are the states of this structure dimension.

Because the number of the inherited attributes of an entity node increases monotonically in the specialization process, the dimension boundary of the other two groups also change with it. The possible range of the space is fixed only after the specialization process, so the specialization state transition must be made before the state transition on the other two groups of dimensions begins. However, the order of transitions between the other two makes no difference to the range of the space.

4.4 The inputs of the *SES-Space*

The inputs of the *SES-Space* must be defined in order for the external world to control the state transition of the *SES-Space*. Based on the states defined in the previous section,

the input primitives can be easily defined. The object-oriented concept must again be observed. All controls to the *SES-Space* use dimensional objects, the pruning nodes, as their management units, so the designation of objects is the first input primitive. The state controls are only made on the designated object. The designated object is called the current pruning node. The basic state transition vectors and the input primitives are listed below:

Specialization State Transition Vectors:

Specialization Select:

To select a specialization node from the specialization group.

Entity Select:

To select an entity node from the entity nodes under the selected specialization node.

These two selections form a layer that can be pushed to the stack.

Aspect State Transition Vectors:

To select an aspect node from the final aspect group.

Variable Value State Transition Vectors:

To set the value to any of the variables in the variable group. The values must be in the ranges of the corresponding variable types.

Multiple Entity Cloning:

To create another pruning-node copy of the current pruning node. It has the same definition node and parent pruning node as the original pruning node does.

It is important that the proper sequence of the specialization state transition and the other two transitions is maintained. The specialization state transition must always be finished first to give the correct ranges of the other two dimensions.

If the above state transitions are positive, the negative transitions also exist. The negative transitions cancel the effect of the corresponding positive transitions. The negative transitions are:

Specialization State Transition Vectors:

The effect of one positive transition can be cancelled when the topmost layer of the stack is popped out. It is important that the specialization group, aspect group, and the variable type group are changed back to the previous ones before the positive transition is made. It is possible that the aspect state is not within the new range of aspect group or the variable type is no longer in the variable type group. These dimensions should either be reset to their initial values, or simply ignored if they no longer exist. When the aspect is already selected and not in the new aspect group, this dimension value should be reset to *not selected*.

Aspect State Transition Vectors:

Delete all child pruning nodes of the current pruning node and unselect the aspect node.

Variable Value State Transition Vectors:

To change the values of the attached variable type group. The values must be in the ranges of the corresponding variable types. Usually, there is only one variable dimension involved each time.

Multiple Entity Cloning:

To delete a pruning node that is created as a clone of the multiple pruning node.

Because of the above negative transition vectors, it is possible to transit from a state to any other states in the *SES-Space*. This capability is important for an interactive system because it allows one to recover from errors. The specialization stack itself keeps the specialization state transition history. It is easy to have a negative state transition vector that rolls back the specialization state to its previous states. For variable value dimensions, keeping the variable transition history is inefficient and not practical. Usually, the history is not kept because of the huge memory needed. If the transition history is not kept, it is difficult to roll back a variable value via its original transition path. Two compromises can be made. One is to offer each variable type an origin. The origin is a reset point once its value transits out of range. The other one is to keep only the last variable value so that the state can be transited back to its previous value.

4.5 The outputs of the *SES-Space*

The outside world is interested in two things from the space. One is the range of the dimensions; the other is the value of the dimensions that represent the position of the state point. The range or the variation of the space is the structural information of the *SES* tree that is always fixed before the pruning process begins. It restricts where a state point can stay. This restriction is enforced throughout the pruning process. A global view and local view of the range of a complex space must be output. The global view is the view of the whole shape of the space. The local view is the space near the current

state point. The global view of a space doesn't need the reference of a state point while the local view is related to the vicinity of a state point. Here, the global view is the *SES* tree itself that includes the information about the structure of the *SES* tree and the hierarchical relationships of the pruning nodes. The local view is about the dimension ranges of the pruning nodes.

Global view:

The following output primitives output the information about the shape of the *SES-Space*. They are not related to the pruning nodes.

Entity view:

Output the attached variable types, specialization nodes, and the aspect nodes under the selected entity node.

Specialization and Aspect view:

Output the attached variable types and the entity nodes under this specialization node or aspect node.

Variable type view:

The ranges of the variable types.

Local view:

The following output primitives output the information about the vicinity of a state point. They are the ranges of the dimensions of a pruning point.

Specialization group view:

The current inherited specialization group defines the range of the specialization process.

Aspect group view:

The current inherited aspect group defines the alternatives of the decomposition process.

Variable group view:

The variable types attached to this pruning node.

Variable type view:

The range of a specific variable type.

The global and local views cover the *SES-Space* range completely. The arrangement of the dimension values of the *SES-Space* is simple. Since each dimension value discussed must be output, the most straight forward arrangement is one-to-one output. Each dimension value has an associated output primitive.

Specialization State View:

To output the stack of layers of current specialization.

Decomposition State View:

To output the current selected aspect node, or none at all.

Variable Type Group View:

To output all types of the attached variables.

Variable Value View:

To output the value of a designated attached variable type.

Pruning Node View:

To output the parent-and-children relationship of a pruning node.

These views cover completely the values of the dimensions. Thus, they allow the outside world to know the correct position of a state point.

4.6 The implementation of the *SES-Space*

All output primitives give both the range of the *SES-Space* and the position of its state point to the external world. Supposedly, the goal state is within the *SES-Space*. There is always a path from the current state to the goal state. Usually, the outside world uses the output primitives to find the direction and distance between the current state and the goal state. It gives the space system the correct direction via the input primitives to reduce the distance. Once the goal state is reached, the purpose of the interaction is fulfilled.

The goal state of an *SES-Space* is the correct pruned structure that stands for a particular task from the given *System Entity Structure* tree. The first step of the pruning process is to output the possible specializations of the current pruning node. The second step specializes the node to a level that fits the particular objective. The last step is to observe all possible decompositions available and decompose the node into a more detailed description of the task until the final task structure is reached. At every pruning node, the attached variables are set to their semantic values according to the application objectives. After all dimensions have obtained their proper values, the goal state is reached. An autonomous approach to reach a goal state has been proposed in [RH87].

Previous implementation of *System Entity Structure* is in PC-scheme and called EPS-scheme [KLCZ90]. Different to the EPS-Scheme, we base our implementation of the *SES*

on the analyzed space dimensions. Once the dimensions, the values, and the input/output primitives are defined, an implementation of the *SES-Space* is straightforward. Basing on the object-oriented programming style, we have the following basic programming objects:

Enode: Entity node, the definition of a pruning node.

ASnode: Aspect or specialization node under an Enode.

Vtype: Variable type that can be attached to an Enode or ASnode.

Pnode: Pruning node of an *SES* tree.

Vvalue: Variable value corresponding to a Vtype. It is attached to a Pnode.

The Enode, ASnode, and Vtype are *SES*-related whereas the Pnode and Vvalue are related to the state of a pruning process.

The Enode has the following slots:

Multiple-Entity-Flag: A flag showing if this Enode is a multiple-entity.

Aspects: A list of ASnodes representing the aspects of this Enode.

Specializations: A list of ASnodes representing the specializations of this Enode.

Attached-variable-types: A list of Vtypes representing the attached variable types of this Enode.

The ASnode is similar to the Enode and has the following slots:

Entities: A list of Enodes representing the Entity nodes under this aspect or specialization node.

Attached-variable-types: A list of Vtypes representing the attached variable types of this node.

The Vtype defines the characteristics of an attached variable:

Type: The special type, such as *integer, real, or string*, of the variable.

Default-value: A default value given for this type of variables.

The Enodes, ASnodes, and Vtypes form a *System Entity Structure* tree. In order for the system to generate a *System Entity Structure* tree which defines the application space, the following primitives must be provided:

Add-Entity: Add an Enode to the *Entities* of an ASnode.

Add-Aspect: Add an ASnode to the *Aspects* of an Enode.

Add-Specialization: Add an ASnode to the *Specializations* of an Enode.

Add-Variable-type: Add an attached variable type to either an Enode or an ASnode.

Goto: Provides the tree traversal capability.

A final system reads in a configuration file that describes a hierarchical application space by using the primitives given above and generates a *System Entity Structure* tree (the *SES-Space*) accordingly. To prune the *SES* tree, we use the Pnodes and the Vvalues to store the state of the pruning process.

The set of slots of a Pnode is more complex than that of an Enode or ASnode. According to the analyzed dimensions of the *SES-Space*, we have the following slots of a Pnode:

Definition-node: The definition Enode of this Pnode.

Aspect-group: A list of all possible decompositions of this Pnode. It is the range of the decomposition-dimension.

Aspect-selection: The selected aspect node. It is the value of the decomposition-dimension.

Specialization-group: A list of all possible specializations of this Pnode. It is the range of the specialization-dimension.

Specialization-path: Current specializations of this Pnode. It is the value of the specialization-dimension.

Attached-variables: A list of actual variable slots corresponding to the inherited variable types. The slots are the variable-type-dimensions.

Parent: The parent of this Pnode in the pruned tree.

Children: The children of this Pnode. Both the parent and children slots are the values of the structure-dimension. The range of this dimension depends on whether the definition node is a multiple-entity and on the aspect nodes of the *System Entity Structure* tree.

In the beginning of each pruning process, a Pnode is created for the root entity of a *System Entity Structure* tree. The implementation of input primitives follows exactly the transition vectors given above for a pruning node. They are:

Select-aspect and Unselect-aspect: Select an aspect of a Pnode. The children of the Pnode are created corresponding to the entities of the selected ASnode. The ASnode must belong to the aspect group of the Pnode. When unselected, all children of the Pnode are deleted.

Specialize and Unspecialize: Add a specialization pair, an ASnode and an Enode, to the specialization path and update the aspect-group and variable-group according to the inheritance axiom.

Set-value and Reset-value: Move the values along the variable dimensions.

Clone-one and Kill-clone: Create or destroy a clone for a Pnode whose definition-node is a multiple-entity.

The tree of created Pnodes is the pruned Entity Structure. The output primitives of an *SES-Space* can be directly implemented by the *get-functions* to instance variables of an object-oriented system.

We have the primitives to create an *SES-Space* and to manipulate its state. Enode, ASnode, Vtype, Pnode, and Vvalue represent the object-oriented dimension-values, and the input/output primitives are implemented as access/query functions to these objects. All values in the Pnodes represent a state of the application space (a task of the application). Access/query functions change the Pnode values to represent another state or task. Because the functions are derived from the input/output primitives, we are sure that all reachable values in the Pnodes completely cover the whole application domain. Thus the functionality of the implementation is ensured. Up to this point, the implementation of a generic application space with its generic input/output primitives is complete.

Each Enode and ASnode has its own label. Multiple occurrences of the same label down the same path produce a recursive path. Multiple occurrences that are not on the same path simply follow the uniformity axiom. It implies that the nodes with the same label are the same Enode or ASnode. We only have to describe a node once. The multiple

occurrences of the node are implemented as multiple links to it to enforce the uniformity axiom. This implementation applies to both the recursive and non-recursive multiple occurrences. As for the pruned entity structure (the Pnode-tree), each Pnode created corresponding to a multiple-occurred Enode has an unique path. The path is created by the applications of the Select-aspect or Clone-one primitives. These two primitives never merge any path of a Pnode-tree whereas the multiple occurrence of an *SES* tree is the merge of two or more paths starting with the nodes of the same label.

We also base our analysis of a mapping model on the dimensions and the input/output primitives implemented for a space. The input/output primitives make the state-space accessible to other models, specifically a mapping model, whereas the function of the mapping model is to map the accessibility to a user. What the user wants to access is the state of the space and how the user accesses the state is by the defined input/output primitives. The mapping model in a control-flow approach links the user to the primitives of this space. If all primitives can be properly linked, the complete accessibility of the user to the space is also ensured. Chapter 6 details how a mapping model links the *SES-Space* to the user and provides the accessibility to the whole *SES-Space*.

The *SES-Space* input/output primitives can also be used as an interface to other computer systems. For example, other computer hosts or even peripherals can interface with an *SES-Space* via these primitives. Because the primitives act as a communication protocol between systems, this kind of interface is much simpler and, instead of complex mapping design, a direct one-to-one mapping to other channels, such as the TCP/IP of computer-networks, is sufficient.

CHAPTER 5

The Visual Tree

The interface channels in the control flow model have only very low-level dimensions, such as the display of a pixel of the video screen or the polling of key presses. These channels are not convenient for programmers to use. Channel buffering provides a higher level of interface for programmers, such as text character manipulation or window management interfaces. Most bufferings provided by the development system are still at a level that is not easy to use for fast prototyping. Visual Tree is a channel buffering technique developed to give the designer very high-level manipulation of the interface channels while still providing the necessary flexibility for most user interface styles. This chapter starts with an introduction to the low-level hardware dimensions. It then discusses the virtual dimensions developed to enhance the channel capability and introduces the visual-tree concept. Visual Tree uses the object-oriented graphic concept to offer a high level manipulation mechanism. It provides a group of virtual dimensions that is much easier to manipulate than are the physical dimensions of the original channels.

5.1 The dimensions of the interface channels

The physical dimensions of the input and output channels are easy to analyze. Each different input or output device has its own operational dimensions. The list here shows the dimensions and the states of most physical devices available.

For input devices:

Keyboard: Each key corresponds to a dimension within the binary range, either pressed or released. The state of the keyboard will be the status of the keys that are pressed and the keys that are not.

Pointing devices: Pointing devices always offer at least two dimensions for positioning. Most pointing devices are also equipped with some on/off buttons. The state of the pointing devices is the physical position or movement and the button states.

Microphone: Provides the dimensions along the audio signals such as the amplitude and pitch of the audio signals.

Sensors: Provide the dimensions along other physical signals such as light, temperature, humidity, etc.

For output devices:

Screen: Each pixel on the screen corresponds to a dimension of either the range of on/off or of a color value. The state is the image shown on the screen.

Speaker: Provides the same dimensions along the audio signals as the input device, microphone. The state is the sound generated from the speaker.

The physical dimensions are always too simple to express high level application semantics. In other words, there is a gap between the physical world and the complex application space. Hardware approaches, such as the increase of screen pixels, the increase of function keys, the increase of mouse buttons and even the introduction of 3-D vision, are employed to extend the number of dimensions. Enlarging the dimensions of the interface channels is always of great advantage to interface the application space and the physical world.

The other approach to extend the dimensions of the interface channels is via software. Software can create virtual dimensions that exist only in the user's conceptual world. The virtual dimension ideas are enforced by using the physical dimensions such as icons on the screen or the earcon[BSG89] on the speaker. Some virtual dimensions associated with their physical devices that are common on today's interface channels are:

- Keyboard: Macro keys, different key combinations.
- Pointing devices: Screen cursors
- Microphone: Voice recognition
- Screen: Window systems, icons, graphic representations
- Speaker: Voice synthesis, the earcons

Virtual dimensions are powerful because they exist in the user's conceptual world. These extra dimensions greatly enhance the power of the interface channels and narrow the gap between the physical world and the application space. The purpose of software channel buffering manipulates the low-level physical dimensions for the programmer and, on the other hand, provides him/her with higher-level virtual dimensions to manipulate.

5.2 Visual Tree dimensions

Visual Tree is a software buffering concept that manipulates the low-level physical dimensions and offers manipulations of a set of high-level virtual dimensions. The manipulation of the virtual dimensions is based on the object-oriented graphics concept. All images displayed on the screen are organized as objects. The object is a group of dimensions that are manipulated as a whole. For example, a window is an object. It is moved, displayed, and erased as a whole.

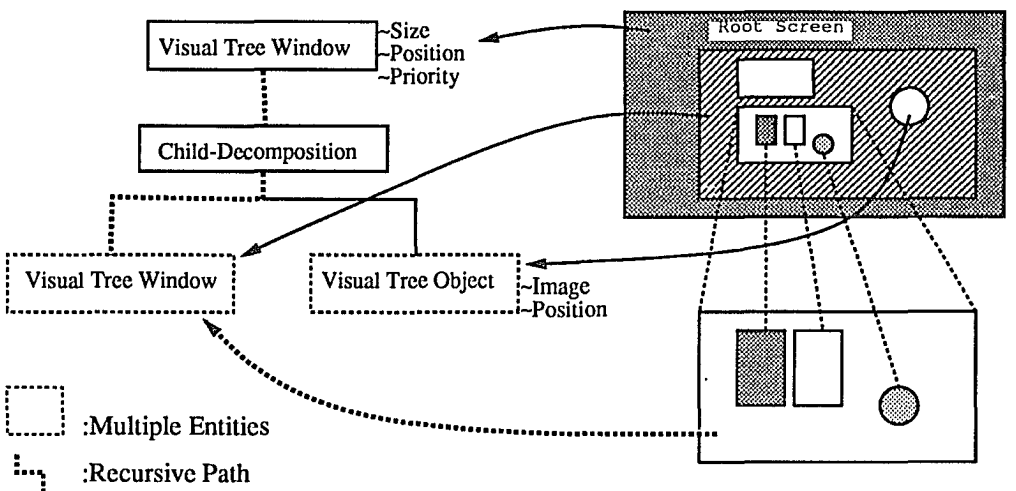


Figure 5.1: The structure of a visual tree described by *System Entity Structure*

As Figure 5.1 shows, the basic *Visual-Tree* nodes on a screen are *Visual-Tree* windows. Windows are generally thought of as areas shown on the video display; but, this is not necessarily so. Windows also exist as areas in memory waiting to be displayed. *Visual-Tree* windows are virtual display screens in the users' conceptual world. Programmers put graphic objects onto those conceptual screens and physical screen images enhance the

virtual-dimension concept. The physical video screen itself is called the root. *Visual-Tree* windows are nodes under the root. They can be put under other *Visual-Tree* windows to form a tree. This tree is called the *Visual-Tree*. *Visual-Tree* objects are the leaf-nodes of the *Visual-Tree*. They are special *Visual-Tree* windows without any children. Usually, the *Visual-Tree* objects are mapped to simple dimension values. For example, the integer and text string values are simple dimension values. *Visual-Tree* windows are used to group the related simple values and make them a manageable unit.

Both sides of the interface channels are now interfacing with each other not through the physical dimensions but the state of the visual tree. The basic structural dimensions of the visual tree are listed below:

- Structural dimensions:
  - The existence of a *Visual-Tree* window
  - The structural location of the window in the visual tree
- *Visual-Tree* window dimensions:
  - The size, shape and image of the window
  - The position of the window
  - The priority of the window

The dimensions on the screen are the virtual dimensions designed for the output channels. Because of the variety of application dimensions, there are windows or groups of windows specifically designed to express certain application dimensions. For example, the *Visual-Tree* object/window for the expression of a truth value can be a button icon and the

object/window for the expression of the value history can be a curve chart. These dimensions are all under the control of a mapping model. Special visual-tree object/windows are discussed in Appendix A.

For the input channels, physical devices such as the keyboards and pointing devices associated with the images on the screen limit the physical dimensions. However, a new virtual dimension comes from object-oriented images on a screen. The value of the dimension is the selection of a graphic object on the screen. The user is allowed to select a certain graphic object from the screen by using a graphic cursor. It is a graphic symbol displayed on the screen and its position is controlled by pointing devices such as mouse/mice, tablet(s) or the cursor keys. The screen is covered by at least a node in the visual tree, so the position of the cursor is covered by exactly one node. The user moves the graphic cursor to a node in the visual tree and performs operations on it by either pressing keys or mouse buttons. The cursor can be in different forms to help the user in perceiving, interpreting and evaluating the system states. No limitations on the number of the graphic cursors is set to provide multiple selections that help the users to specify and execute the actions required. Input dimensions on the designated windows are the physical clicking of the keys or buttons. Clicks can be further decomposed into several stages:

Press: Pressing a key downward.

Hold: Holding down the key.

Release: Releasing the key.

Usually, the keys include mouse buttons and keyboard keys. Additional virtual input dimensions can be created from variations of the operations on the keys. In the Visual Tree environment, the virtual dimensions provided are:

Mouse Buttons:

Single click: Press and release immediately

Double click: A sequence of two single clicks

Drag: A sequence of pressing, holding, moving, and releasing

Keyboard Buttons: The combined operation of several keys and buttons.

Note again that all input dimensions are associated with graphic windows or objects. The drag operation further allows the user to control the object dimensions of the *Visual-Tree* and provides direct manipulation of the user interface[Shn83].

There are virtual input dimensions associated with special windows. The virtual input dimensions have values meaningful only to the special object/window dimensions. For example, string editing is associated only with the text editing window. They extend the physical dimensions and the control power of the users.

5.3 The basic primitives of the Visual Tree

The Visual tree is highly object-oriented; everything is managed as a *Visual-Tree* window/object. Different groups of dimensions are handled in separate ways. For the structure dimensions, a tree manager is in charge of all updates of the structure; for example:

- Add a *Visual-Tree* window to a parent
- Delete a *Visual-Tree* window from a parent
- move a *Visual-Tree* window from one parent to another

For object dimensions, each window object is in charge of its own dimensions.

- Move a *Visual-Tree* window to a new position
- Change the size of a *Visual-Tree* window
- Change the shape of a *Visual-Tree* window
- Hide or show a *Visual-Tree* window
- Other special *Visual-Tree* window output dimensions

As for the input channels, each input event has the following basic dimensional values:

- The graphic window selected
- The button values via the virtual dimensions: 1-click, 2-click, hold and release. ...
- The keyboard values via virtual dimensions: the ASCII code or special key codes.
- The special window input dimensions: values of special variable types

An input event is a value set of input virtual dimensions and can be used as the base for other higher level virtual dimensions such as the input command languages. Input sequences from high level languages can be parsed and translated into low level semantic primitives.

Different window systems need different implementations to realize the Visual Tree concept. A *Visual-Tree* implementation on a TI-explorer window system is given in Appendix A.

CHAPTER 6

The Design of Mappings

Same semantic meanings have different representations in different worlds. For example, an on-off state of a light-bulb in a computer is of binary form while it is the actual light emitted in the physical world. Mapping is to transform different representations from one world to another. This chapter introduces the generic mappings of an *SES-Space* onto *Visual-Tree* virtual dimensions, but first consider the following ideas used in the design sequence.

6.1 Several considerations

The mappings decide the outlook of the system and thus define the system image explained in the chapter on the control flow model. The user's mental model can be loosely defined as the model that the user applies in predicting the behavior of the interactive system. In other words, it is the user's view of the system image. Different people could have very different mental models. The user manual and the system image provide an intended mental model for intended users. The actual mental model is formed when the user uses the system and learns the application space and the mapping space through the user manual. The intended user's mental model might deviate from the real mental model

of a user because of the incompleteness of the manual, the insufficient understanding of the manual, and different system usage patterns. The gap can be narrowed if the mapping style is clear. As mentioned in Chapter 2, there are three types of mappings involved in the mapping model. The user interface style guides the dimension mappings; the multiple mappings offer different ways to achieve the same space input and makes the user interface more flexible. Designing a more complete set of mappings will provide a more powerful user interface, yet covering all possibilities will not be efficient. Selecting only the needed mappings is important for fast prototyping. In the *SES-Space*, each dimension is equally important and all should be accessible to the users, so there is little need for level mappings. Some complex variable types such as the position and size need some level mappings, though, to give users a high-level interaction with grouped values instead of with individual values.

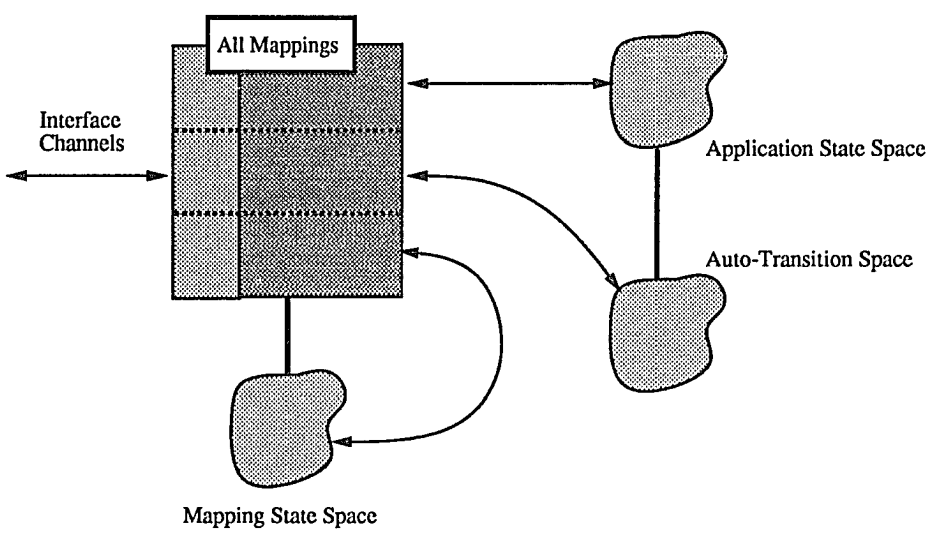


Figure 6.1: The three different state spaces

There are other state spaces to be mapped besides the *SES-Space*. They are the state spaces of the mapping model and the auto-transition model (Figure 6.1). These state spaces represent the system from a different perspective. Each space is mapped onto a distinctive group of interface channel dimensions to help the user manage the different aspects of system dimensions.

Four basic elements of each dimension must be mapped:

- The dimension direction
- The value or state
- The value range
- The transition vector

The first three elements are for the outputs of the dimension and should be mapped onto the output channels whereas the transition vector control is input and should be mapped onto the input channels. The combined appearance of the first three elements helps the user perceive, interpret, and evaluate the current state of the system. When the user is familiar with the direction and the range of the interesting dimension, such as the text screen of an editor, the first two elements can be hidden from the user to save precious interface channel dimensions for other mappings. The example in Figure 6.2 shows how a scrolling bar gives the user the directions and the range of a given dimension. As for the mappings of the input transition vectors, they usually appear as the input commands of the system. The commands can be either the combination of key sequences or the movements of the mouse and the clickings of its buttons. The virtual dimensions for inputs can be infinite if a computer language is provided. The grammar of the language

also can be complex enough to provide a natural language interface[Ric84], though the mappings are difficult to design. Most users adapt to simple mappings for inputs more easily than complex mappings.

Usually, if users are familiar with the mappings, including the inputs and outputs of a space and the use of the interface channels, they are regarded as experts. To make a novice user become an expert fast, the mappings should be designed as simple as possible and in a style that the user can get familiar with easily. Proper mappings of the four basic elements are very important.

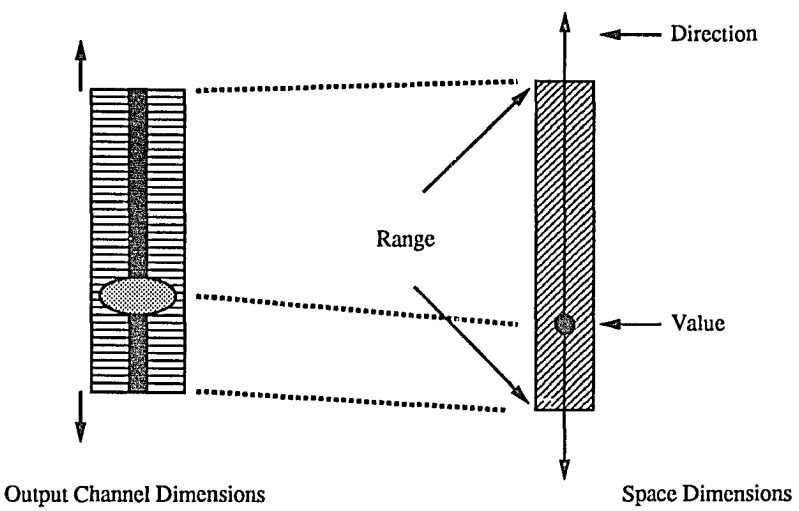


Figure 6.2: Direction, range and value of a dimension

Direct manipulation[Shn83] is a user interface style that puts both the state transition vectors and the outputs, especially the outputs of the dimension values, onto the same group of virtual dimensions. Virtual dimensions available for direct manipulation are limited, mainly the existence, the shape, the position and the size of a virtual output object,

such as the structure and object dimensions of the *Visual-Tree*. To a direct-manipulation user interface style, the input operations should associate with these limited virtual dimensions. The linkage between the graphic object and the input events provides exactly the needed information for direct manipulation. The most common direct manipulation style is to map a dimension onto the 2-D space of a graphic object. That is, the value is mapped as the relative position of an object, the range is mapped as a bounded area. The position of the object controls the input of the dimension. In Figure 6.2, the relative button position represents the value within the range of the dimension. The controlling of the value is via the movement of the button position along the direction. Note that the movement of the object is always combined with the object itself representing the value and creates the direct manipulation style.

Physical operations on a key are simple. To preserve the simplicity for the user in the stage of specifying and executing actions, the input language should be kept simple. Because of the simple input language used, the input virtual dimensions are limited. It is impossible to map all the inputs of a space onto the limited dimensions. Input modes or command hierarchy is used to solve the problem. The mode and the hierarchy increase the complexity of the mapping space and take away the advantages of simple key operations. Care must be taken to minimize the number of interface modes and the command hierarchy that contributes additional complexity to a mapping space.

Another mapping idea is object-oriented mapping. The object-oriented mappings give high level management of the related dimensions. The group of the dimensions that are put into an object should be expressed on the output channels as a graphic object

to convey the physical object perception; the *Visual-Tree* window is an example. Also, the object-oriented mappings support the direct-manipulation interface style. When a group of dimensions are mapped onto a graphic object on the screen, values of grouped dimensions are displayed as the position, size, and the image of the graphic object. Modifying these values is to change the position, size, and the image directly. This gives the direct-manipulation flavor.

6.2 The design convention

For the consistency of the user interface style, similar application dimensions should be mapped onto similar virtual dimensions or even the same dimension. Consistency reduces the time spent by the user in specifying actions to be performed, because the user needs less time to induce the similar mapped actions. Several conventions are designed to enforce consistency.

Object-oriented style:

All dimensions are grouped as objects. The objects are mapped onto the output channels as a graphic object and are always kept together. Inputs to the same dimensions are also made via the same graphic objects.

Direct Manipulation:

The state transition vector is mapped onto the same virtual channel dimension that the transited value is mapped onto. Because the object-oriented interface style groups the outputs and the inputs together in an object, the direct-manipulation technique is straightforward in such an environment.

Hierarchical dimension management:

All dimensions are managed hierarchically to help the user in handling the complex space dimensions and to provide the designer with a natural mapping style to the limited output channel dimensions.

Hierarchical traversal:

The operation of traversing the dimension hierarchy should always be the same. Each level of the hierarchy is mapped into a *Visual-Tree* window with the children representing the lower level abstractions. Input operations on the *Visual-Tree* window are for the traversal of the dimension hierarchy.

Value view and setting:

The dimensional value should be output with its direction and range. The operation of setting a value should always be the same. The values are always mapped as the leaf nodes in a dimensional hierarchy tree. Operations on the leaf nodes are the setting of the values.

Separation of the application and the mapping space:

Because of the different semantic meanings, the dimensions of the application and mapping spaces should be mapped onto different *Visual-Tree* windows to clearly

distinguish one from another for the user. Mapping effects accessible to the users, though, should be mapped closely to the effect itself; for example, the scrolling bar of a window should be attached to the window itself.

Simple mouse button operations:

Mouse button operations are limited to the ones provided by the *Visual-Tree* environment. No other new operations are introduced. Designers should use these simple button operations only and avoid complex operations.

Consistent display of available user operations:

The display should express clearly the kinds of operations allowed. Because button operations are limited, legal operations on a graphic object should be displayed if possible. This will help the user in specifying the correct actions.

According to the above conventions, all possible basic input operations are grouped. As mentioned, the buttons on a mouse can have three basic operations in the *Visual-Tree* environment; the single-click, double-click, and the hold and drag. Multiple buttons cause confusions for the users. In the design of this generic user interface, all three buttons are made to have the same function to simplify input operations as in the Macintosh environment. Hold and drag is suitable to change a value continuously; it is assigned to change the value of any dimensions. To set discrete values, the single-click is the simplest one. Sometimes when an object is associated with two or more inputs, the double-click operation can be used. For complex or precise value inputs, the keyboard is used. In such a case, a certain level of familiarity with the mapping space is required.

6.3 Dimension Mappings of an *SES-Space*

The *SES-Space* is object oriented and can be easily arranged into hierarchical relationships as shown in Figure 6.3. This arrangement offers hierarchical abstraction of the space dimensions.

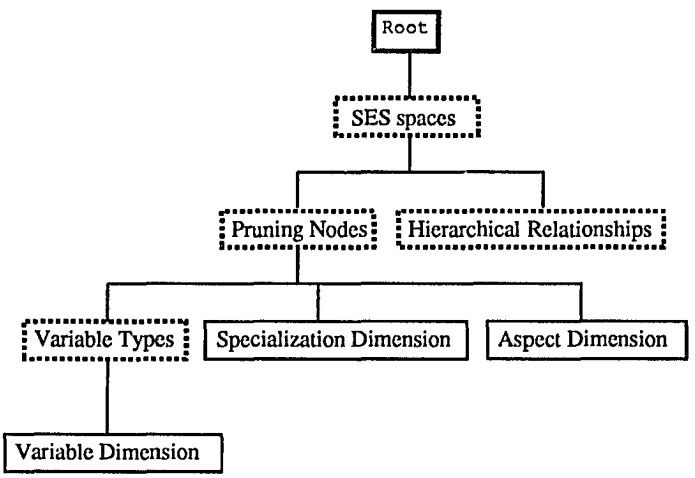


Figure 6.3: The hierarchy of the *SES-Space* objects

Eventually, all dimensions must be accessible by the users. Because of the limited available output dimensions, only some of the dimensions can be mapped at a time. By using the hierarchical arrangement of the dimensions, the user can choose the suitable level of space-dimension abstraction. The hierarchy-traversal operations of such an arrangement are inevitable in order to access all dimensions.

We arrange the dimensions of the *SES-Space* into a hierarchical tree. The root node is the software system itself with several *SES-Spaces* available. Under each *SES-Space*, we have pruning nodes and the hierarchical relationships among these pruning nodes. Each

pruning node has its associated specialization, aspect, and variable type dimensions as described in Chapter 4. To access a particular dimension, one has to traverse down the hierarchy but the complexity of the operations is limited because of the relatively few levels of abstraction.

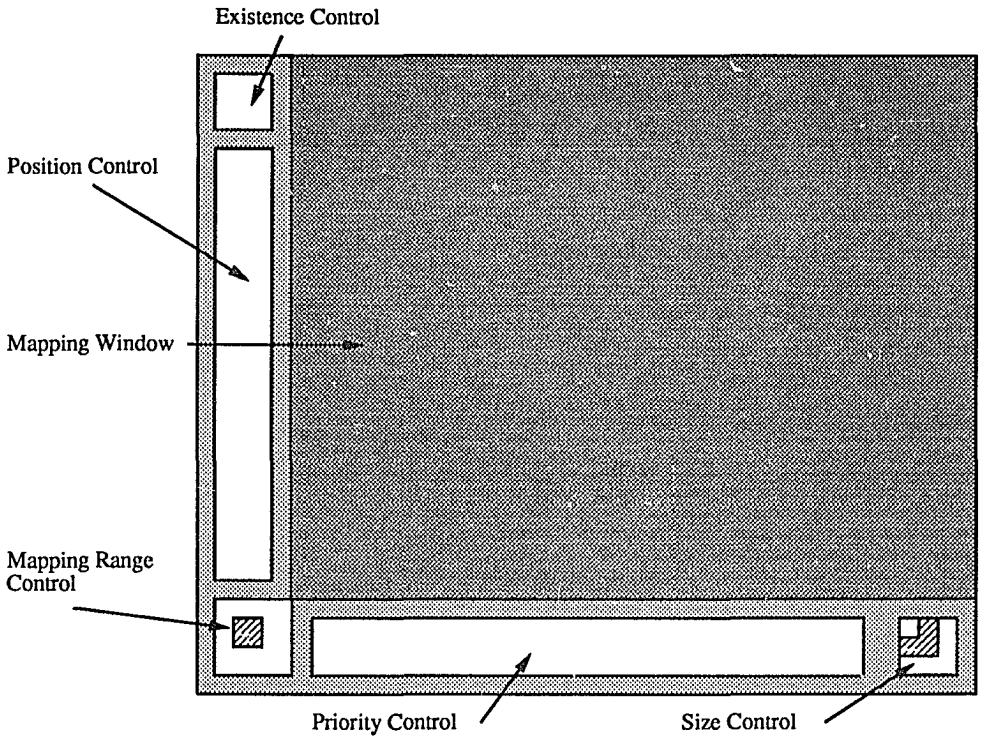


Figure 6.4: The control of the dimensions of a mapping window

The *Visual-Tree* window is a good management unit for a dimensional object. A mapping window is a *Visual-Tree* window mapped to a certain group of dimensions. The dimensions, such as their existences, sizes, positions and mapping ranges, of some mapping windows are controllable by the user. These dimensions are related to the mapping space and not the application space. They should be mapped onto different objects or windows

according to the convention used. Currently, the common style is to display the mapping status and the available controls just around the mapping windows (Figure 6.4). If a control is not available, it is not shown. A single-click on the existence control deletes the mapping window. Windows under a parent can overlay with each other. The topmost one has the highest priority. Users should be able to control these priorities in order to bring the most interesting window to the top. A single-click on the priority control raises or lowers the window's priority. A hold and drag on the position control, size control and range control can move, change size, and change mapping range of the mapping window, respectively. The mapping range is a new dimension created due to the limited space of the mapping window. Sometimes, it is impossible to map all the values of the group of the dimensions onto a single mapping window simultaneously and only part of the values are mapped. The users can always control the mapped range and, thus, the value of this range dimension.

Each level of dimension hierarchy is naturally mapped to a mapping window. The direct mapping from the dimensional hierarchy to the *Visual-Tree* hierarchy is not needed, though. A direct hierarchical mapping will force all parent levels of current window to exist as the parent *Visual-Tree* windows. This mapping style conveys object orientation. However, not all parent windows are of interest to the user; they occupy screen area and waste channel dimensions. Furthermore, the hierarchical abstraction levels are very simple and the traversing path is not that important. Therefore, another approach is to make all windows appear at the same *Visual-Tree* level as do the mappings of the Macintosh File Finder.

The traversal of the dimension hierarchy is to create the corresponding mapping window. Common user interface design maps the creation of the mapping window to the double-click on the corresponding graphic object in another mapping window. This tradition is followed in this design and the single-click is used for other purposes. A single-click on the existence control of a mapping window deletes the window.

6.4 The root level of abstraction

The topmost level of dimension abstraction is the system information about the available *System Entity Structure* trees and the result of the previous pruning processes. There are two kinds of information stored in separated group of files. One is the *System Entity Structure* tree file and the other is the pruned tree file. The pruned tree file is always associated with a *SES* tree file as the definition file. The state of this dimension is the existence of these files. The transition direction is the creation, deletion and the naming of the files. The range is not bounded and, therefore, is not mapped to the output channels.

The most straightforward and common way nowadays in mapping the existence of these files is simply the display of the file names in a list. The state of the mapped range is the portion of the list displayed. Its 1-D transition direction is the choices of the portion of files displayed. The range is bounded by the existing files. This mapped range dimension maps onto the 1-D range control as shown in Figure 6.5.

To traverse another level of abstraction, simply double-click on a file name to create the mapping window for the saved *SES-Space* or pruned tree.

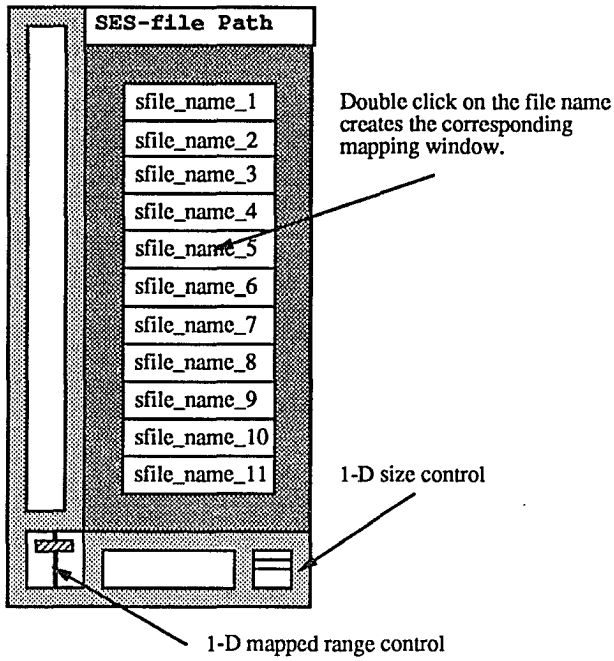


Figure 6.5: The mapping window of the root

6.5 The *SES-Space* level of abstraction

Two groups of dimensions in the hierarchy are to be displayed on this level of the mapping window. They are the pruning nodes and the structural relationships of these pruning nodes.

Usually, the relationships also represent the hierarchical decomposition of a node. They can be displayed on the mapping window in several ways as shown in Figure 6.6. Each display method has its advantages. The indented tree and the graphic tree are similar; they enable one to see a hierarchical structure directly in a tree form though the graphic tree offers better presentation. Usually, when each tree node has more than two children,

the width of a tree will grow faster than its depth. For example, if each node in the tree has two children. The width of the tree will be $2^{depth-1}$. In a graphic tree, the width of the tree goes horizontally in the same direction as the length of the text labels goes. It produces an image that is less compact than the indented tree whose width goes vertically in the different direction from the length of the text labels. Thus, the indented tree is more screen efficient than the graphic tree.

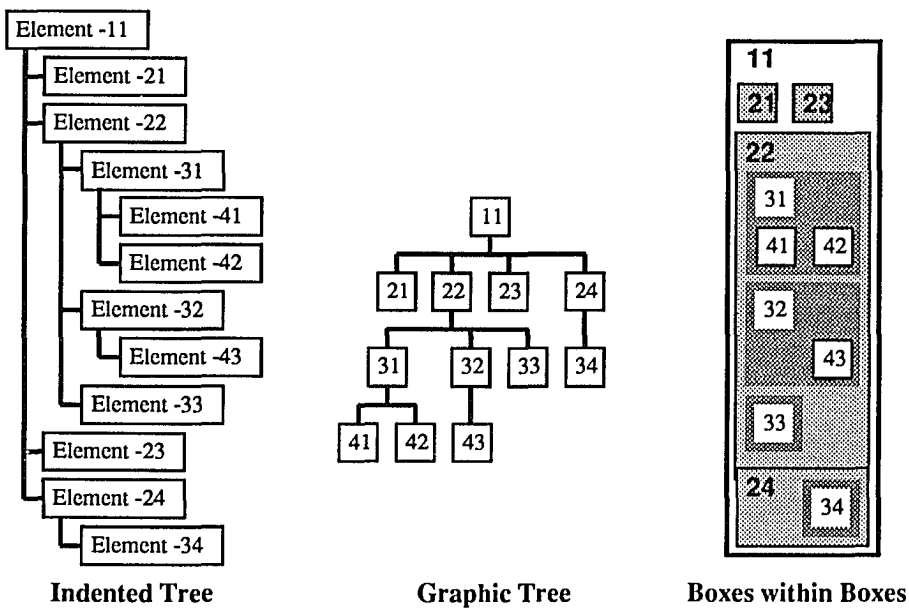


Figure 6.6: The three types of the hierarchical display

In the boxes within boxes style, once the hierarchical level grows too deep, the boxes create confusion and make it more difficult to compare the level differences between the boxes. The relative position of the boxes makes this style more suitable in mapping the sibling relationships in the same level. For example, the coupling relationships in the models of Discrete Event Simulation[Zei84] can be displayed nicely in this style.

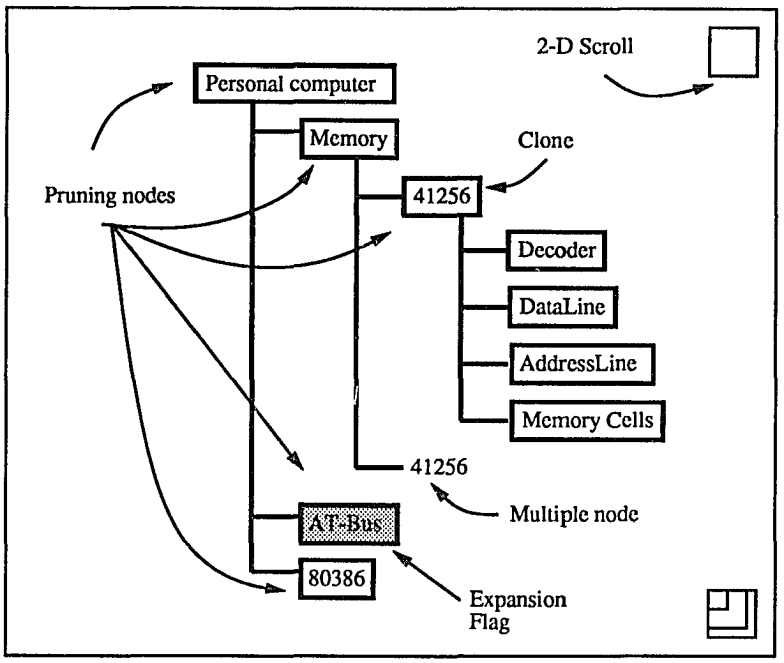


Figure 6.7: The mapping window of an *SES-Space*

As Figure 6.7 shows, the style adapted in this design is the indented tree that offers the best screen efficiency. The pruned tree is displayed to a level of detail controlled by a user. Each boxed label represents a pruning node. If the node is expanded, all its children are displayed. If not, an expansion flag is attached to show that its children are not displayed. Multiple entities are not boxed while their clones are boxed by solid-lines. Frequently, the tree is too big to fit in the mapping window. The mapped range dimension also exists in such an occasion. There are two ways to control the range of the mapped portion. One is to control via the hierarchical structure. That is, the user can decide whether the child nodes under a parent node are displayed or not. If the child nodes are displayed, the parent node is *expanded*. All nodes with an expanded parent are displayed on the

mapping window. If the expanded tree is still too large to fit in the mapping window, the panning of the tree on the mapping window is controlled via the 2-D mapped range control (scroll control).

Recall the convention that a double-click traverses a tree hierarchy. Thus, a double-click traverses the dimension hierarchy from a pruning node to its next level; its specialization, aspect, and variable types and invokes the creation of another mapping window for the pruning node. The single-click can be mapped naturally to toggle the expansion of the node.

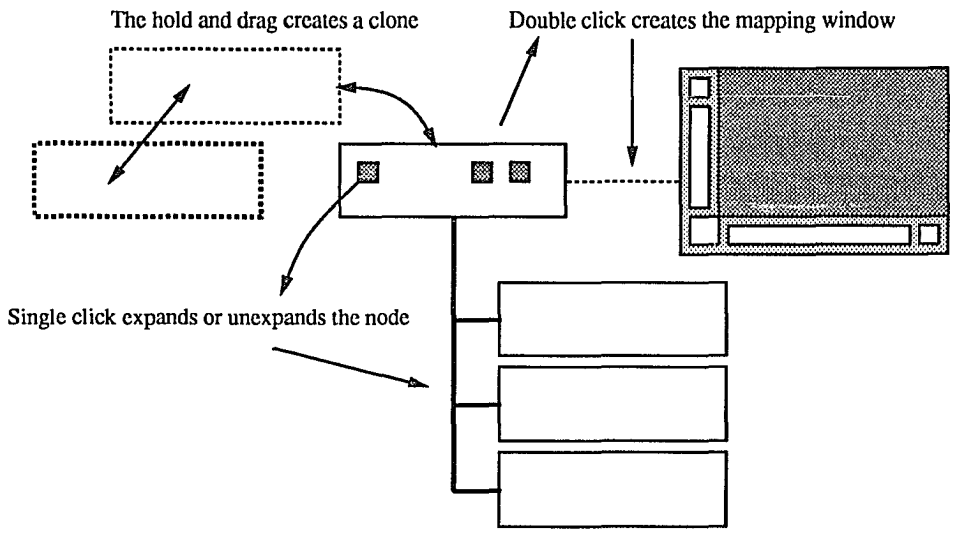


Figure 6.8: The operations on a pruning node

In this level of abstraction, there is one state transition vector for an application dimension. The dimension is the existence of clones for multiple entities. Because single- and double click operations are already assigned to the range control of the mapped dimensions, the available operation left is the hold and drag. As Figure 6.8 shows, holding

and dragging a multiple entity template creates a clone of the entity at its side. Holding and dragging a clone back to the entity deletes the clone.

6.6 The pruning node level of abstraction

The main elements to be mapped at this level are specialization, aspect dimensions, and attached variable types. Because of the great variety of variable types, their individual dimensions are pushed down to a deeper level. As a convention, a double-click on a variable type creates the mapping window that corresponds to the dimension of this variable type. The mappings of this level are shown in the Figure 6.9.

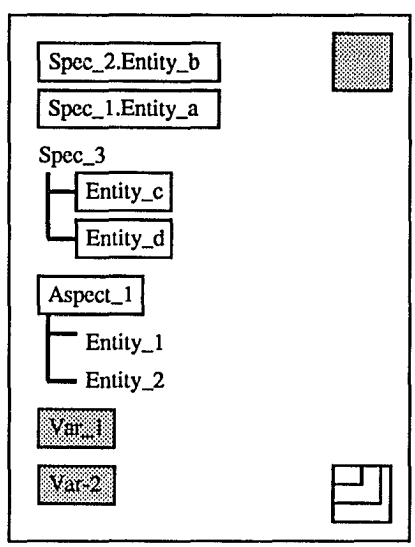


Figure 6.9: The mapping window of the pruning node

The selections of a specialization and the aspect values are the inputs involved in this level. The selection of a specialization and aspect is made through a single click on the

node. To undo a specialization is to click on the last selected specialization on the stack. To undo an decomposition is to single-click on the selected aspect.

As the groups of specialization, aspect, and the number of the variable types are frequently too large to fit in the mapping windows, the new mapped range dimensions are created in these cases and are mapped onto the 1-D mapped range controls.

6.7 The attached variable level of abstraction

The last level of mapping is the value of a variable type. Each variable type defines a dimension with its own state, transition direction, state range, and transition vector.

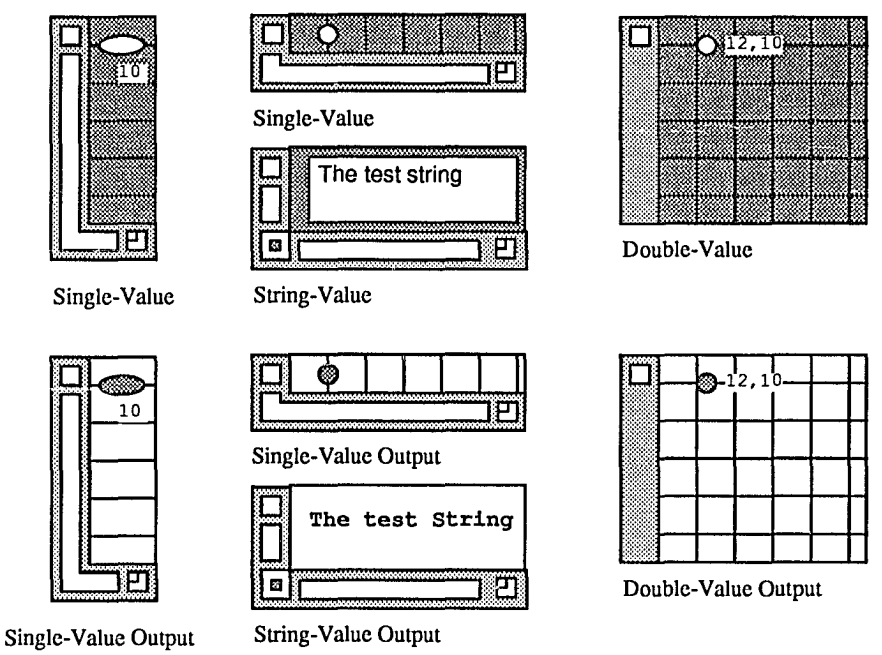


Figure 6.10: The mapping windows of some basic variable types

Following the previous conventions, several basic variable types are provided in the *Visual-Tree* environment as Figure 6.10 shows. Some variables are for output only. They are useful when the dimensions are of interest to the user but are not accessible. These kinds of dimensions are common in control and monitor systems where the values are not controlled by the user but by sensors; for example, the temperature values controlled by the temperature sensors. Mapping windows having only outputs should be different from the ones with both inputs and outputs; for example, shaded icons are for output only. Because of the direct manipulation style, windows with only input do not exist.

Most typed values map to the size and position of an icon. The text string dimensions are much more complex than simple variable types. Direct manipulation of text variables follows the conventions of available text editors.

6.8 Other mappings

All *SES-Space* dimensions described in Chapter 4 are mapped properly and consistently. For some variable values, they are significant not in the lowest but in a higher abstraction level. For instance, an entity has position and size as its attached variable types. It is better to map these values to the position and size of the entity graphic object icon directly so that the user needs not traverse to the lowest level of abstraction. This higher level of mappings is even more important when there are sibling nodes with related attached variable types. Namely, the relative positions and the sizes of sibling nodes are

very important semantically. To provide the flexibility of other output and input mappings, software interface hooks must be provided. The parent node of the siblings is a good place to place the hook.

The software display hook is placed in each pruning node where the decomposition information (the children) is kept. A generic display routine is hooked to a pruning node; it is in charge of the outputs of the node's children. The pruning node expansion is under the control of the generic display routine.

Other facilities, such as the locating of the interesting dimensions, can help the users select the right mapping result and reduce the time spent in user activities. They include searching, matching, and the keeping of the traversal history. These operations are complicated and it's not suitable to map them to simple button operations. Operation modes are inevitable. The designer should always provide hints for possible follow-up operations to the user when an operation mode occurs.

CHAPTER 7

Examples

In this chapter, we give three examples to illustrate how to generate a generic user-interface for a hierarchical control system. They are user interfaces of a smart-home control system, an *SES* editor, and a DEVS modeling environment.

7.1 Smart-home Control System

The Smart-Home control system allows a home-system programmer to manage control and data-acquisition units in a hierarchical manner and to program the smart behavior – the rules of a house.

7.1.1 *SES* description

First of all, we must describe the smart-home control system in *SES*(Figure 7.1) hierarchically.

A home or house can be decomposed into several manageable units. Garden, recreation area, building area, and patio are examples of such units. Each unit has a unique name and geometric information. For example, a garden is named *Backyard Garden* and its geometric information is kept, as boundary lines, in the attached variable:Unit-Geometry.

Down the hierarchy, big units are decomposed into smaller sub-units. The sub-units can be specialized into area, floor, room, or misc. An *Area* is to provide another level of management and introduces a recursive structure to this system. The recursiveness allows a home to be described to any level of detail via decomposition.

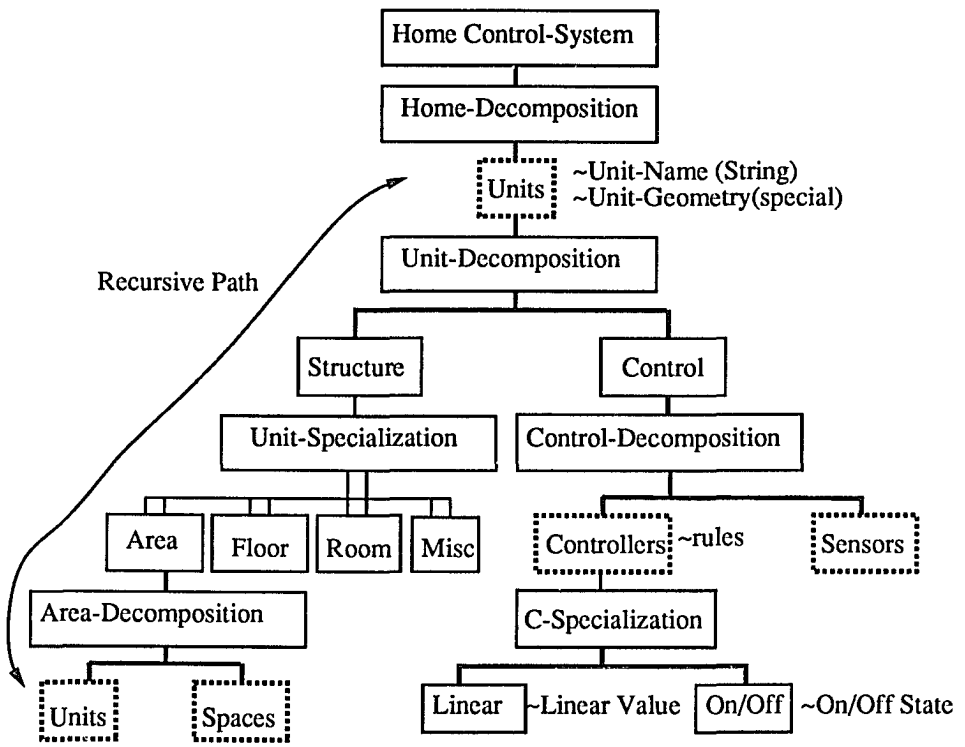


Figure 7.1: The *Recursive System Entity Structure* of a home-control system

As for the control-portion of the unit-decomposition, there are physical controllers and sensors. Sensors gather data from the physical world. The data include temperature, humidity, noise-level, phone-status, and etc. The attached variable: sensor-link of each *Sensor* entity is linked to a sensor that sends back physical world information. On the other hand, the controller is specialized into either a linear controller or an on/off switch

with their own attached variables, linear value and an on/off state, respectively. Obviously, the data retrieved from the sensors can only be output; however, for the controllers, a user not only can *see* the settings but also *modify* them.

Smart features of this home-control system come from attached rules of a controller. A rule decides *how* a controller acts on certain conditions. The conditions are based on the information in the hierarchy. Not only data from sensors, but also data of the structure of a home can be used. Actions are carried out by an inference engine based on an assigned algorithm. Most actual actions carried out are on/off or adjusting operations of controlled switches. A home-system designer writes default rules and the user modifies the rules to suit his/her personal needs. Because the inference engine is an autonomous part of the system, users usually need not interface with it. If we want a user to control the behavior of the inference engine, another child under *Home-Decomposition* called inference-engine should be added to describe its application space.

7.1.2 Mapping design

We see from the types of the attached variables that some of them are application-oriented. For example, *Unit-Geometry* and *Setting-points* are special types. Developing under the *Visual-Tree* environment, it is easy to implement the mapping objects/windows of these special-typed variables. One could also decompose the special variables to a lower-level of only generic variable-types like strings, integer or real numbers. For each of the attached variables, we design a special semantic window. Figure 7.2 shows some possible special *Visual-Tree* windows.

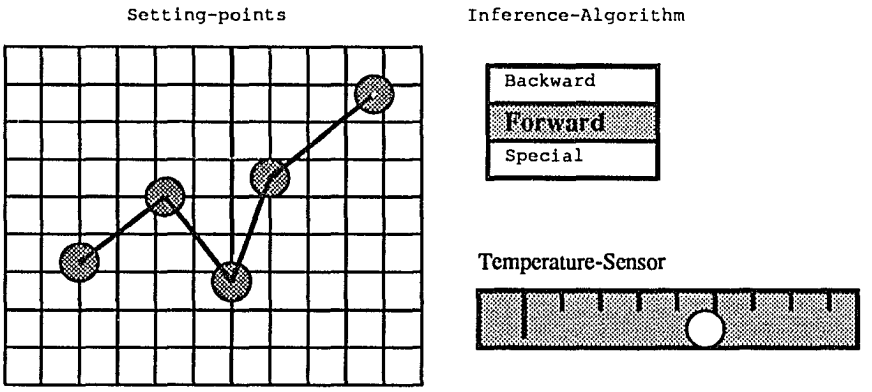


Figure 7.2: The windows for the special variables

Due to the application oriented aspects, we want the geometric boundaries displayed among siblings to be in one level. A special mapping method is hooked to the entity of *Unit*. The method displays boundaries of the children of a unit in a map like style as in Figure 7.3.

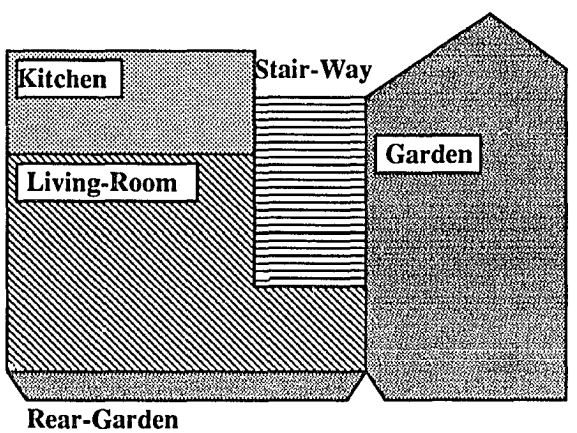


Figure 7.3: The boundary-like unit-level mapping

All user-interface conventions described in Chapter 7 are enforced in this design. For example, a double-click operation is always linked to a traversal in dimension hierarchy.

7.1.3 The configuration file of a smart-home control system

All designs in previous sections are put in a configuration file that includes an *SES* describing the system and the special mappings. First, we preload the special windows/objects and methods implemented in the last section, and then we feed the generic user-interface system with a configuration file describing this user-interface so that it can be generated accordingly.

The configuration file of the smart-home control-system is listed below:

```
;;-------------------------------------------------
;; a configuration file to generate the user interface
;; for a smart-home control-system
;; the sequence is important due to the tree traversal
;; the format is
;; (command-key label <parms>)
;; (:up)
;; (:new-pes)
;; (:new-aspect name)
;; (:new-specialization name)
;; (:new-entity name multiple-p?)
;; (:new-ses name structure-mapping dimension-mapping)
;; (:new-varaibel-type name def-value
;;                     (vmapping (op arg) (op arg)))
;; :new-ses must always be the first one to generate
;; the root of the system entity structure
;;
(:new-ses "Home Control System"
;;
;;        the default structure presentation vtree window
          structure-default
;;        the default dimension presentation vtree window
          dimension-default)
(:new-aspect "Home Decomposition")
(:new-entity "Unit" t)
(:new-variable-type "Name"
                    "Unit-name"
;;                  a generic variable mapping window
                    vil-generic-variable-map)
(:new-variable-type "Geometry"
```

```
                    '((10 10) (20 10) (20 20)
                    (10 20) (10 10))
                    vil-generic-variable-map)
;;-----------------------------------------------
(:new-aspect "Unit Decomposition")
(:new-entity "Structure Part")
(:new-specialization "Unit Specialization")
(:new-entity "Area")
(:new-aspect "Area-Decomposition")
;; the beginning of the recursiveness
(:new-entity "Unit")
(:up)
(:new-entity "Space" t)
(:up)
(:up)
(:up)
(:new-entity "Floor")
(:up)
(:new-entity "Room")
(:up)
(:new-entity "Misc")
(:up)
(:up)
(:up)
(:new-entity "Control")
(:new-aspect "Control Part")
(:new-entity "Controller" t)
(:new-variable-type "Rule"
     ()
     vil-generic-variable-map)
(:new-specialization "Controller-type")
(:new-entity "On/Off")
(:new-variable-type "on/off"
;;    vil-switch: special variable (on/off) mapping window
  nil (vil-switch))
(:up)
(:new-entity "Linear")
;; a special variable value
(:new-variable-type "Linear"
;;               vil-dial: a special variable mapping window
                    20 (vil-dial
                          (:set-width 100)
                          (:set-height 200)
                          (:set-style :Y)
```

```
                        (:set-ylow 10)
                        (:set-yhigh 200)
                        (:set-yresolution 30)
                        (:set-ylabel-resolution 4)))
(:up)
(:up)
(:up)
(:new-entity "Sensor" t)
(:new-variable-type "location"
                    "location"
                    vil-generic-variable-map)
(:new-variable-type "comment"
                    "comment"
                    vil-generic-variable-map)
(:new-variable-type "sensor-value"
                    35 (vil-dial
                        (:set-width 200)
                        (:set-height 100)
                        (:set-style :X)
                        (:set-xlow -10)
                        (:set-xhigh 100)
                        (:set-xresolution 10)
                        (:set-xlabel-resolution 5)))
;; last statement create the root
;; of the pruning nodes
(:new-pes)
```

7.2 An *SES* Editor

According to Figure 3.4, we write the corresponding *SES* language in the configuration file.

7.2.1 The configuration file of the *SES* Editor

```
(:new-ses "Entity"
        structure-default
        dimension-default t)
; alternating mode from entity to aspects
; and specializations
(:new-aspect "Entity-to-Specs-Aspects")
```

```
(:new-entity "Aspect" t)
(:new-variable-type "Name"
                    "Aspect"
                    vil-generic-variable-map)
(:new-variable-type "Coupling"
                    "Coupling"
                    vil-generic-variable-map)
; alternating mode from aspect to entities
(:new-aspect "Aspect-to-Entities")
(:new-entity "Variable-type" t)
(:new-specialization "Variable-specialization")
(:new-entity "Boolean")
(:up)
(:new-entity "Integer")
(:up)
(:new-entity "Real")
(:up)
(:new-entity "Text")
(:up)
(:new-entity "List")
(:up)
(:new-entity "Position")
(:up)
(:new-entity "Generic")
(:up)
(:new-entity "Function")
(:up)
(:up)
(:up)
(:new-entity "Entity" t)
(:up)
(:up)
(:up)
(:new-entity "Specialization" t)
; alternating mode from specialization to entities
(:new-aspect "Specialization-to-Entities")
(:new-entity "Variable-type" t)
(:up)
(:new-entity "Entity" t)
(:up)
(:up)
(:up)
(:new-entity "Variable-type" t)
(:new-pes)
```

Entity-to-Specs-Aspects, *Aspect-to-Entities*, and *Specialization-to-Entities* enforce the alternating mode axiom. The recursive path starts from the root *Entity node* which is a multiple-entity. However, the root is not allowed to be cloned even though it is a multiple-entity. This is a restriction to ensure one single *root*. The variable specialization provides all kinds of variable types to be used.

7.3 A DEVS Modeling Environment

Two basic models used in a DEVS Modeling Environment are coupled-model and atomic-model. Atomic-models are leaf nodes in the hierarchy while coupled-models are non-leaf nodes. The recursive path is the one through coupled-model to the lower DEVS-Model. The *System Entity Structure* is shown in Figure 7.4.

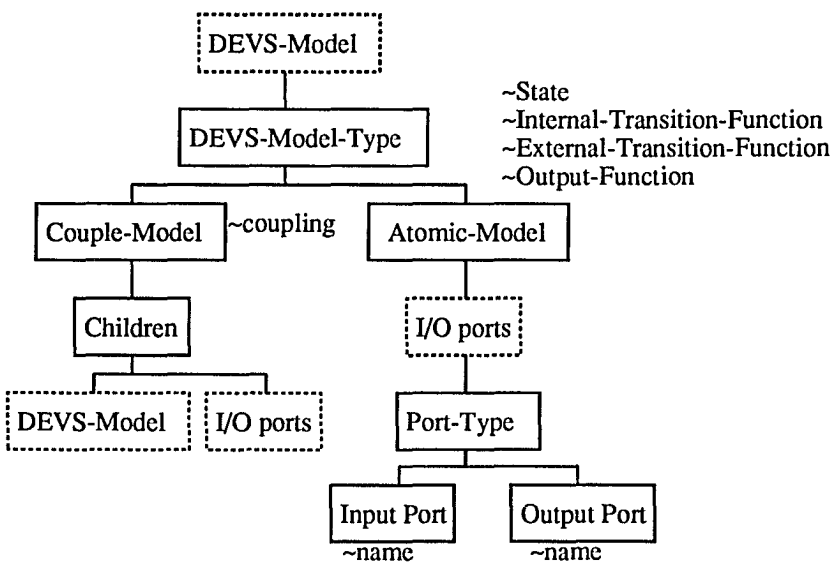


Figure 7.4: The *Recursive System Entity Structure* of DEVS Modeling Environment

The most special attached variable is the coupling of Couple-Model. A special mapping window must be designed for it to give users of the modeling environment a proper virtual dimension(Figure 7.5).

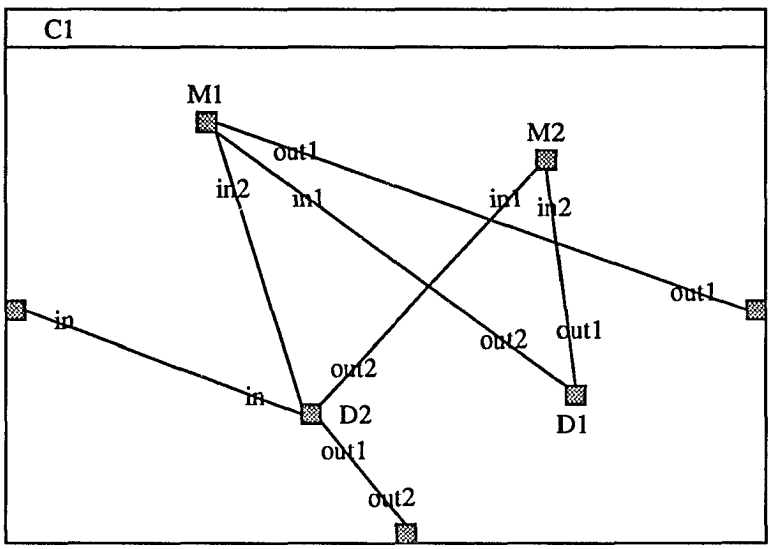


Figure 7.5: The mapping window of a coupling variable type

7.3.1 The configuration file of the DEVS Modeling Environment

```
;;-------- Devs related --------------------------
(:new-ses "Devs-Model"
    structure-default
    dimension-default t)
(:new-specialization "DEVS-model-type")
;;----  under devs-model-type  ----
(:new-entity "Atomic-Model")
(:new-variable-type "Internal-transition-function"
        '() vil-generic-variable-map)
(:new-variable-type "External-transition-function"
        '() vil-generic-variable-map)
(:new-variable-type "Output-function"
        '() vil-generic-variable-map)
(:new-variable-type "State"
```

```
    "Passive"
    vil-generic-variable-map)
(:new-variable-type "Sigma"
    0
    vil-generic-variable-map)
(:new-aspect "Atomic-model-decomposition")
;; -- under atomic-model-decomposition
(:new-entity "I/O port" t)
(:new-specialization "Port-type")
(:new-entity "Input Port")
(:new-variable-type "Name"
    "Port-name"
    vil-generic-variable-map)
(:up)
(:new-entity "Output Port")
(:new-variable-type "Name"
    "Port-name"
    vil-generic-variable-map)
(:up)
(:up)
(:up)
;; -- under atomic-model-decomposition
(:new-entity "State-variables" t)
(:new-specialization "Variable-type")
(:new-entity "List")
(:new-variable-type "List"
    ()
    vil-generic-variable-map)
(:up)
(:new-entity "Symbol")
(:new-variable-type "Symbol"
    "Symbol"
    vil-generic-variable-map)
(:up)
(:new-entity "Integer")
(:new-variable-type "Integer"
    0
    vil-generic-variable-map)
(:up)
;;-- variable-type --
(:up)
;;-- state-variables --
(:up)
;;-- Atomic-model-decompostion --
```

```
(:up)
;;-- Atomic-Model --
(:up)
;;-- Devs-model-type --
(:new-entity "Coupled-Model")
; this is an important variable in devs environment
; it should be specifically given a mapping window
(:new-variable-type "Coupling"
    '()
    vi-generic-variable-map)
(:new-aspect "Children")
(:new-entity "Devs-Model" t)
(:up)
(:new-entity "I/O port" t)
(:new-pes)
```

CHAPTER 8

Conclusion

The control flow model of the interaction mechanism gives the designer a predictive model to evaluate design decisions. It reduces the chances of an iterated design process and thereby the design cost. It also takes the designer through a systematic design sequence; the application space analysis, the interface channel space analysis, and the mapping space design. The activities in the user model affect most of the design decisions.

The design process applies the dimensional space concept to manage the complex design domain, such as the application, interface channels and the mapping design. The space dimensions are the interactive elements of an interactive system.

In the analysis of hierarchical systems, the space dimensions are derived from the axioms and the pruning process of the *System Entity Structure* language. The *System Entity Structure*'s descriptive ability covers hierarchically decomposable systems and manages them into space dimensions.

To interface the physical world with the application space, the *Visual-Tree* environment is developed. It provides a high level interface environment for both the user and the system designer. The system designer maps the low-level application space primitives to the high-level interface-objects on the *Visual-Tree*. The user manipulates the objects via

the assigned physical operations of the input devices. This environment greatly reduces the implementation time and makes user interface styles more efficient.

Applying both the analysis of the *SES-Space* and the *Visual-Tree* virtual dimensions, the design of the mapping links between the two is implemented. Design decisions are based on the results predicted from the effects on the stages of users activities. The mappings result in a generic user interface that is easy to learn and efficient to use. Full access to the application dimensions ensures the functionality of the implemented system.

Compared to other user interface management systems such as TAE plus [Cen88] and InterViews [LVC89], the uniqueness of the *SES* user interface management system is that it not only generates a user-interface but also the application space as well. Most user interface management systems try to separate themselves from the development of application systems and lose the links between the user interface and the functionality of the application spaces. However, the *SES* user interface design utilizes the common characteristics of hierarchical systems to simplify the design of both the application system and its user interface. The design process is now as simple as writing a description of the intended application in the *System Entity Structure* language. The generated user interface is generic and easy to customize. The user interface consistency is also ensured by the pre-defined *SES-Space* dimensions that exist in all hierarchical applications.

There are a few areas that are not explored by this dissertation. Because of the variations between the hierarchical application, the generic user interface cannot satisfy some crucial aspect of the user interface. The software hooks on the crucial dimensions

are essential. Efficient user interface styles of these different application domains must be studied.

The mapping designs should be different between novices and experts. The full power of the input devices is not explored. It weakens the interaction efficiency of the experts. Multiple mappings such as hot keys are the solution.

Grammatic input languages increase the expressive power of expert users, but they are too difficult for novice users. Their expressive power includes mapping facilities such as searching and grouping of dimensions. On the other hand, if a task can be expressed via programs or scripts of a certain language, it can be solved analytically in batch mode instead of user interactive mode. Usually, an application system should provide both modes to suit different needs.

The *SES-Space* dimensions can also serve as an interface protocol between computer systems. Further study of communication efficiency via this protocol should be studied especially when the interface is made on a busy network channel.

In this dissertation, the control flow model only provides a qualitative prediction relating to a design decision. Quantative analysis is also possible to provide more precise predictions. More statistical data about the effects on the user activities must be collected before quantative prediction is possible. Evaluations of a user interface are possible if its design can be described in terms of the control flow model.

Future studies include research on different application spaces and the search of their optimal mapping styles. Often, combining many optimal mappings doesn't lead to a

global optimal performance; usage patterns of the user do not necessary follow the design's original intention. Further study of user's behavior, pertaining to different mapping combinations, is important. Simulations of a user interface style and its efficiency are possible by using the control-flow model and the gathered statistical data of user activities. The simulations will lead to precise predictions and efficient system design.

Appendix A

Implementation of Visual Tree on TI Explorer

TI Explorer is a symbolic machine running Common-Lisp [Tex87a]. It offers an object-oriented Lisp called Flavor; its graphic window system[Tex87b] is built on this Flavor system and designed to support a great variety of window applications. Its low-level graphic primitives are complete and support different user-interface styles. However, it lacks a consistent design methodology. A consistent design methodology guides a programmer through a clear and short design path and leads to a successful user-interface implementation. The graphic primitives of the Explorer window system are at a level that is too low for fast-prototyping. Programmers must combine many primitives to achieve a task at a higher semantic-level much in the same way as programming in assembly language. High programming-levels are very essential for fast-prototyping. The Explorer window system does provide some high-level window flavors, such as text-edit windows, scrolling-bars, and query windows, that are easy to use; however, their user interface styles are out-of-date, and sometimes difficult to use.

The *Visual-Tree* concept offers a more consistent user interface and an easier to use programming environment to a programmer than the Explorer window-system. *Visual-Tree* provides high-level virtual dimensions, such as dials, toggle-switches, and curves,

to users. Users deal only with direct and consistent virtual dimensions in their mind. Programmers operate on fixed-format messages passed from a *Visual-Tree*. For example, the value of a variable "temperature setting" is represented by a dial-like image (the virtual-dimension). The temperature dial is a *Visual-Tree* object that provides consistent and easy operations to users. Once put on a *Visual-Tree*, it passes back fixed-format messages that can easily be manipulated by a programmer(Figure A.1).

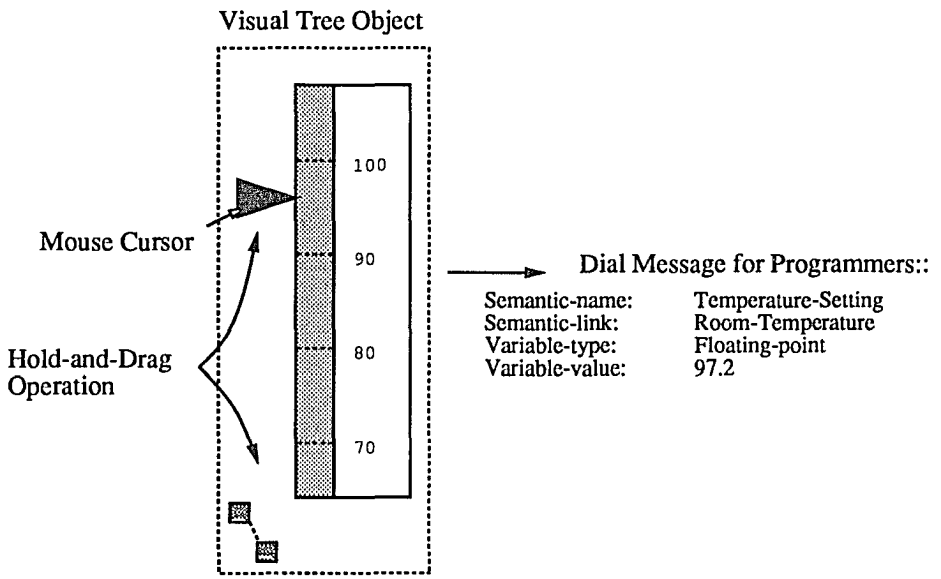


Figure A.1: A dial object of Visual Tree

To apply the *Visual-Tree* concept to an Explorer window system, we have to meet the following requirements:

Input-event management: Input-events from a user must be collected, dispatched, and translated properly into high-level messages.

Message management: High-level messages must be consistent and easy to access from a programmer's view point.

Visual-Tree objects: *Visual-Tree* objects are linked to variables representing application dimensions. They provide virtual dimensions at a level that is comfortable to users. The virtual dimensions and the control operations must be consistent and easy to use.

Visual-Tree windows *Visual-Tree* windows arrange related *Visual-Tree* objects into manageable groups. They are important to virtual dimension management. The programming of the windows should be simple.

Section A.1 explains the input-event and message management. Sections A.2 describes the implementation of *Visual-Tree* windows and objects. Section A.3 provides the user interface design methodology in a *Visual-Tree* environment.

A.1 *Visual-Tree* input-event and message management

The basics of a *Visual-Tree* environment is that it provides end-users with a consistent and high-level operating style and translates the operations into high-level messages that a program can easily handle. *Visual-Tree* deals, for programmers, with all low-level input operations. Usually, the Explorer window system handles mouse or keyboard inputs and dispatches them toward individual windows. In the *Visual-Tree* environment, all *Visual-Tree* windows/objects are under a root window. Similar to the X-window model[Tex89], inputs are channeled through the root (X-window server) and dispatched

to a window/object (X-window client). The window/object, receiving the dispatched inputs, translates the inputs to messages for programmers.

The *Visual-Tree* hierarchy provides a logical way to dispatch the input-events according to the visibility of windows. Observing the overlaying windows in hierarchy, the following rules define the visibility of a window.

- All windows are opaque. If a window is *covered* by another window. The covered portion is not visible.
- A parent window is always *covered* by its children (windows or objects); the children are opaque so the covered portion of the parent is not visible to users. Due to limitations of the Explorer window system, the screen area of a child can never exceed the area of its parent.
- Siblings of the same parent window are stacked according to their priorities. Windows/objects that have higher priority *cover* those with lower-priorities.

The mouse cursor, controlled by a pointing device, always stays on one of the visible or partially visible windows. The window that the cursor stays on is the window that receives inputs. Based on the same idea, the Explorer window-system provides a routine, associated with the mouse-controlled graphic cursor, called:

```
;; explorer routine
w:window-owning-mouse
```

The following routine returns the visible window/object pointed to by the graphic cursor in a *Visual-Tree*.

```
;; visual tree implementation
object-owning-mouse
```

In contrast to the *Visual-Tree* environment, the Explorer window system never sends input information to partially visible windows. We, thus, have to implement a mechanism that allows a partially visible window to receive inputs.

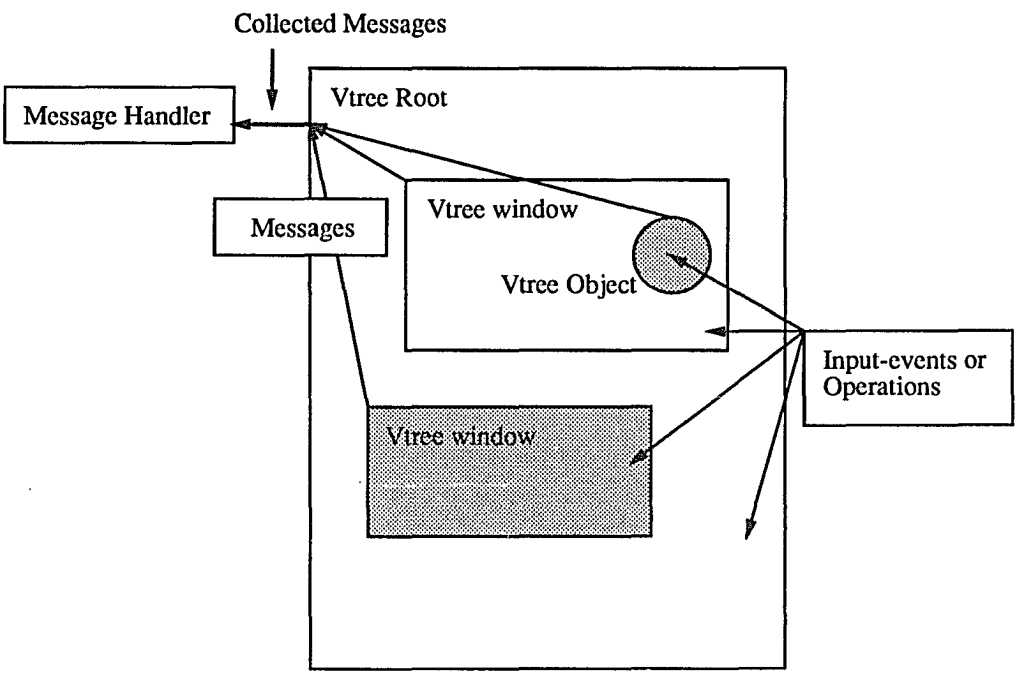


Figure A.2: The programming model of a *Visual-Tree*

Visual-Tree usually uses the screen as the root. In the TI-Explorer environment, the screen is shared by many other windows such as the LISP-Listener, the Text Editor, etc. that don't behave like *Visual-Tree* windows. To avoid conflicts of mouse handling between Explorer windows and the *Visual-Tree* windows, a root *Visual-Tree* window is

specifically created for each *Visual-Tree* environment as a child-window under the *w:main-screen* [Tex87b]. The root is responsible for input dispatches to all its children when it is *selected*[Tex87b]. Once the mouse-cursor stays on a window/object's visible portion, the root dispatches inputs to it. When the window/object receives the dispatched inputs, it translates the inputs into consistent messages that are easy to handle by a programmer.

All *Visual-Tree* windows are implemented as Explorer windows. Similar to the window-functions and the application event-queue in the Microsoft window system[Mic90], available channels for other program modules to receive messages are the message-passings between objects and the input buffer of each window. To provide a consistent and clean interface channel between a *Visual-Tree* and other program modules, only the root window is used(Figure A.2).

If we use only the input-buffer of a root, the interface is even cleaner; however, the input buffer delays the response time for a message to be handled by other processes. We leave a link between a *Visual-Tree* and other modules; the *handler*. If the instance variable: *handler* of the root is nil, the root will put the collected messages into its own input buffer, else it will send the messages to the handler that has a method called *:handle-message*. The response time is faster by using :handle-message because everything is running on the same process. To allow a root to collect messages, all windows/objects send messages to their :alias-for-inferiors (that is its parent) until the root is reached. The following methods are used in an Explorer window system and our *Visual-Tree* implementation:

```
:alias-for-inferiors
; when a message is generated, simply send it to
; the :alias-for-inferiors of this window.
```

```
; the :alias-for-inferiors of a root is itself
; for other windows, is the parent

:force-kbd-inputs
; A root uses this method to queue the messages
; received in its input buffer.
```

Input-event and message handling are defined in the flavors called *vwindow*, *vobject*, and *vroot*. A programmer must mix these flavors to whatever windows/objects he/she creates in a *Visual-Tree*. The *vroot* is a special flavor that enables a root to dispatch inputs and collect messages. The following *Visual-Tree* methods are used for message interfaces within the window hierarchy. All windows/objects that generate special messages must redefine these methods to give correct behavior.

```
:mouse-enter
; when the mouse-cursor entering this window/object

:mouse-leave
; when the mouse-cursor leaving this window/object
; usually, a message is generated at this instance

:mouse-moves x y
; x, y is the cursor position relative to
; the upper-left corner of the window/object
; Whenever a mouse-cursor is under the control of
; a window/object, this method responses to the
; mouse movements.

:mouse-clicks x y click-number
; x, y is the cursor position relative to upper-left
; corner of the window/object.
; whenever a click or a series of clicks is recognized,
; this method will be called

:mouse-hold x y
; when the mouse button is held down for a while
; and recognized not to be a click
```

```
:mouse-holding x y
; when the mouse button is still held,
; this method is called periodically

:mouse-release x y
; when the mouse button is released

:mouse-standard-blinker
; It sets the default graphic cursor
; for the mouse on this window/object.
```

The consistent format of a message delivered to a root window is as followed:

Semantic-name: A special name of the window/object. It is useful in deciding semantic meanings of an operation to a window.

Semantic-link: A link to a mapped application-dimension that is related directly to the above semantic-name.

Variable-type: The type of the mapped application-dimension.

Variable-value: The new value of the mapped application-dimension.

Each window/object can generate its messages at any time within any interfacing methods according to the need of the virtual dimension it represents. Whenever there is a need to interface with an application dimension, there should be a message generated for a program to apply the state transition vector on the dimension. The programmer should know the special message-formats of windows/objects used before any implementation. He/she can also devise his/her own window/objects as long as the correct flavors have been mixed and the proper programming-interface methods are defined. The consistency of the user-interface style must always be followed.

For a mouse, only the following operations are regarded as meaningful:

- Single-Click
- Double-Click
- Hold-and-drag

A window/object handles one or more styles of operations. The window/object should clearly indicate what operation(s) are available. The indication method of operation styles available is always the same in the *Visual-Tree* environment as shown in Figure A.3. Flavors that handle only one style of the operation are also implemented as *mixins*.

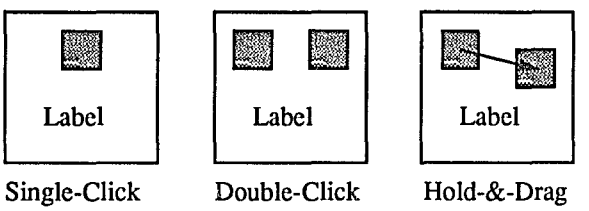


Figure A.3: The three *Visual-Tree* operation-indicators

A.2 *Visual-Tree* Windows and Objects

Visual-Tree windows are implemented as Explorer windows except for the way in which they handle the mouse. The *Visual-Tree* root stops the original Explorer mouse-process and creates its own mouse-process whenever it controls the mouse. The process handles *Visual-Tree* input-events and dispatches them into the correct window via mouse-enter, mouse-leave, mouse-moves, mouse-click, mouse-hold, mouse-holding, and mouse-release methods.

Interfacing methods are defined for all generic *Visual-Tree* windows. Window operations, including closing, positioning, resizing, and raising windows, are provided by

special mixin-flavors that ensure a consistent user-interface style. *Visual-Tree* window flavors, such as *vwindow*, must be mixed in first for proper operation. Special behavior can be defined for *Visual-Tree* windows by overwriting these methods or mixing in with new flavors.

In the *Visual-Tree* hierarchy, the *Visual-Tree* objects are the leaf-nodes of a tree. Though they are conceptualized as special *Visual-Tree* windows, we didn't implement them as windows in the Explorer system for the following reasons:

- The overhead of a graphic object is far less than that of a window; *Visual-Tree* objects need not handle children and all the graphics capabilities that *Visual-Tree* windows do.
- The purpose of a *Visual-Tree* object is different from a window; a *Visual-Tree* window is mainly used as a tool to organize *Visual-Tree* objects and does not have special dimension mappings while a *Visual-Tree* object maps an application dimension to a virtual dimension.
- The TI-Explorer system excludes several visual effects of *Visual-Tree* objects, such as overlaying images, if they are implemented as windows.

As mentioned earlier, the purpose of a *Visual-Tree* object is to map an application dimension to a virtual dimension. The mapping is bidirectional; one direction is from a user operation to a state transition vector and the other is from the state of a dimension to the external world. In Chapter 6, we showed that there are four elements to be mapped for each dimension.

- The dimension direction
- The value or state
- The value range
- The transition vector

Mapping a value onto a virtual dimension is called output. Mapping an operation to a transition vector is called input. The other two elements, direction and range, are the reference frames for the direct-manipulation user-interface style. A *Visual-Tree* object must present all four elements. The basic implementation is the reference-frame flavor. On top of the reference frame, output and input flavors are implemented. Different types of *Visual-Tree* objects have different requirements for their reference frames. For example, in a 1-D dial, its reference frame has the following attributes:

```
Instance-variables
parent                ; the parent vtree window
semantic-name         ; the name of this object
semantic-link         ; the application dimension
low                   ; lower bound of the value
high                  ; upper bound of the value
position              ; position of the dial in a window
size                  ; size of the dial in a window
tick-length           ; length of a tick
tick-resolution       ; value resolution between ticks
orientation           ; horizontal or vertical
label-font            ; font of tick-labels
label-format          ; format of tick-labels
label-resolution      ; number of ticks per label

; methods associated with reference frame
:draw                 ; draw the frame on parent window
:erase                ; erase the frame from the window
```

The output flavor of a 1-D dial has the following attributes:

```
Instance-variables
value                ; the dimension value
indicator-font       ; graphic-cursor definition
indicator-char
indicator-offsetx    ; x offset of the cursor
indicator-offsety    ; y offset of the cursor
value-font           ; font of the value label
value-format         ; format of the value label
value-position       ; position of the value label

; methods associated with the output flavor
:update              ; update the value image
```

Two methods of the input flavor of a 1-D dial translate user's input-operations into a value. The input-operations are dispatched to the 1-D dial by :mouse-moves, :mouse-hold, :mouse-holding and :mouse-release methods.

```
Instance-variables
trace-font           ; tracing-blinker definition
trace-char
trace-offsetx        ; the offsets of the blinker
trace-offsety
value-type           ; the type of the value

; method associated with the input flavor

:mouse-enter
; . create a special blinker for this dial
; . set the standard mouse blinker to it

:mouse-hold
; . changes the mouse-blinker to a 1-D blinker

:mouse-holding
; . moves the tracing blinker

:mouse-release
; send the root with a message about this dial
```

```
:mouse-leave
; . changes the mouse-blinker back to original blinker
```

Combining the above three flavors, a programmer can create a dial for both input and output or for output alone. A programmer simply creates a 1-D dial object with the correct initializations and adds it to its parent; Set the *superior* instance variable of an object and send an :activate message to it to make it one of the children of the *superior*. The programmer then can receive its messages from the root window.

An object library of many simple dimensions, such as toggle switch, integer, floating point, double integer, double floating point, value history, label-selection, etc., is implemented on the Explorer window system with complete comments. Their programming interfaces are clearly commented in [Cho90].

A.3 *Visual-Tree* design methodology

Visual-Tree objects reduce the time needed for a programmer to implement mappings between application dimensions and virtual dimensions. *Visual-Tree* windows help a programmer manage the dimensions. A programmer must always analyze the involved application dimensions and group the dimensions in a logical way that can be directly mapped to *Visual-Tree* windows. After correct *Visual-Tree* window/object assignments to the application dimensions, the implementation of interface channels is done.

What is left for the implementation is to translate the messages passed from the root into the correct state transition vectors. Because the messages are at a level that is close to the application dimension, the translation is direct and easy.

Here is an example to illustrate the *Visual-Tree* design methodology. Let's say there are two rooms, A and B. Room A has three lamps with on/off switches. Room B has two lamps with variable-brightness controls. To design a control system for these two rooms, we have to analyze the application space first.

We represent the on/off switches by boolean values and the variable controls by two floating-point variables ranging from 0 to 10. On the actual brightness of the lights, we assign a feedback sensor for each light and link it with a variable called "brightness". These are the only application dimensions involved in this example. We assign each dimension a proper *Visual-Tree* object. For example, the boolean value is assigned a toggle-switch and the variable control is assigned a 1-D floating-point dial. The toggle-switch is given a semantic-name, light-switch, and linked to one of the lights in Room A. Similarly, the 1-D dial is given a semantic-name, light-control, and linked to one of the lights in Room B. Each feedback sensor is assigned an output-only *Visual-Tree* object.

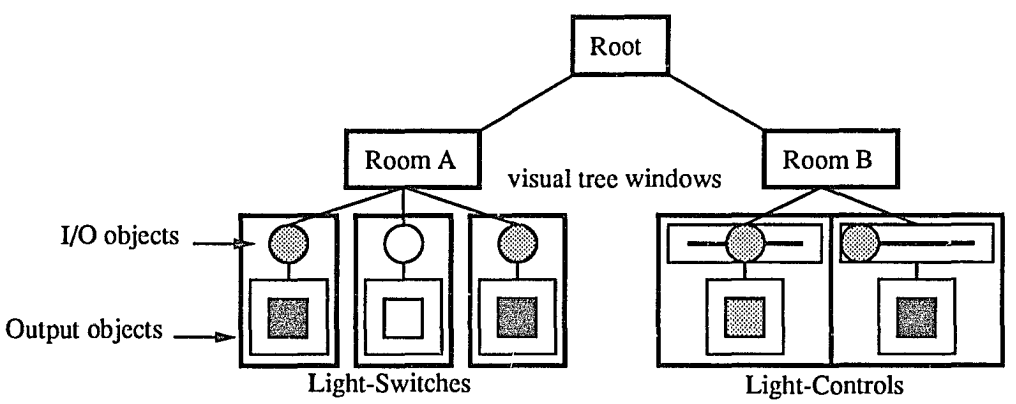


Figure A.4: The *Visual-Tree* of the light-control system.

After the *Visual-Tree* object assignment, we can further group the application dimensions so that they are easy to manage. The logical way is to group the control and the sensor of each light as a unit. Then, put the lights of the same room in one group and we assign each unit and group a *Visual-Tree* window. Its *Visual-Tree* hierarchy is shown in Figure A.4.

Next, we translate the messages received from the root into actions.

```
loop:
  case semantic-name:
    light-switch:
      ;; the control of the state transition vectors
      turn semantic-link to the message value
      ;; output from the application space
      update the sensor output of the semantic-link
    light-control:
      set semantic-link to the message value
      update the sensor output of the semantic-link
    vtree-window:
      ;; control of the mapping space
      case message-value
        close
          close semantic-link (simply delete the window)
        size
          sizing semantic-link
        ...
        ...
      end case
  end case
end loop
```

For complex systems, simple-dimension *Visual-Tree* objects are not sufficient. Manipulations of *Visual-Tree* windows/objects in the library are needed to create special virtual-dimension behaviors. Nevertheless, the simple and straightforward design methodology always makes the life of a programmer easier and also the user-interface consistent.

REFERENCES

[BB87] Ronald M. Baecker and William A. S. Buxton, editors. *Readings in Human-Computer Interaction: A multdisciplinary approach.* Morgan Kaufmann, Los Altos, California, 1987.

[BKP90] Susan Bovair, David E. Kieras, and Peter G. Polson. The acquisition and performance of text-editing skill: A cognitive complexity analysis. *Human-Computer Interaction*, 5(1), 1990.

[BSG89] Meera M. Blattner, Denise A. Sumikawa, and Robert M. Greenberg. Earcons and icons: Their structure and common design principles. *Human-Computer Interaction*, 4(1), 1989.

[Car84] S. K. Card. Human limits and the vdt computer interface. In John Bennett, Donald Case, Jon Sandelin, and Michael Smith, editors, *Visual Display Terminals: Usability Issues and Health Concerns.* Prentice-Hall, Englewood Cliffs, N.J., 1984.

[Cel86] Francois E. Cellier. Combined continuous/discrete simulation – applications, techniques and tools. In *Proceedings of 1986 Winter Simulation Conference*, pages 24–33, Washington, D.C., 1986.

[Cen88] Century Computing, Incorporated, Laurel, Maryland. *Introduction to TAE PLUS, Version 3.11*, 1988.

[Cho90] Chung-Hen Chow. *Visual Tree Source Code.* Knowledge-based System Design and Simulation Lab, University of Arizona, Tucson, Arizona, 1990.

[CMN83] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates, Hillsdale, N.J., 1983.

[CWZ90] Francois E. Cellier, Qingsu Wang, and Bernard P. Zeigler. A five level hierarchy for the management of simulation models. In *Proceedings of 1990 Winter Simulation Conference*, 1990.

[FvD82] James D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1982.

[GL85] J. D. Gould and C Lewis. Designing for usability: Key principles and what designers think. In B. Shackel, editor, *INTERACT '84: First conference on human-computer interaction*, Amsterdam, 1985. North-Holland.

[HK88] Scott E. Hudson and Roger King. Semantic Feedback in the Higgens UIMS. *IEEE Transactions on Software Engineering*, 14(8):1188–1206, August 1988.

[KLCZ90] Tag Gon Kim, Chilgee Lee, Eric R. Christensen, and Bernard P. Zeigler. System Entity Structure and Model Base Management. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(5), Sep./Oct. 1990.

[KP83] David E. Kieras and Peter G. Polson. A generalized transition network representation of interactive systems. *Proceedings of the CHI'83 Conference on Human Factors in Computing*, pages 103–106, 1983.

[KZ89] Tag Gon Kim and Bernard P. Zeigler. Hierarchical scheduling in an intelligent environment control system. In *The Second International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert System*, Tullahoma, Tennessee, 1989. University of Tennessee Space Institute.

[LVC89] Mark A. Linton, John M. Vissides, and Paul R. Calder. Composing User Interfaces with InterViews. *Computer*, 22, Feb. 1989.

[Mic90] MicroSoft Corporation, Wedmond, WA. *Microsoft Windows Software Development Kits; A guide to Programming. Version 3.0*, 1990.

[ND86] Donald A. Norman and Stephen W. Draper, editors. *User Centered System Design: New Perspectives on Human-computer Interaction.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986.

[Nor83] Donald A. Norman. Design principles for human-computer interfaces. In *Proceeding of CHI'83*, pages 1–10, 1983.

[NS79] William M. Newman and Robert F. Sproull. User interface design. In *Principles of Interactive Computer Graphics*, chapter 28. McGraw-Hill, New York, 1979.

[RH84] Richard Rubinstein and Harry Hersh. *The Human Factor: Designing Computer System for People.* Digital Press, Burlinton, MA, 1984.

[RH87] Jerzy W. Rozenblit and Y. M. Huang. Constraint-driven generation of model structures. In *Proc. of the 1987 Winter Simulation Conference*, pages 604–611, Atlanta, 1987.

[RHKZ90] Jerzy W. Rozenblit, Jhyfang Hu, Tag Gon Kim, and Bernard P. Zeigler. Knowledge-based design and simulation environment (KBDSE): Foundational concepts and implementation. *Journal of the Operational Research Society*, 41(2):101–114, 1990.

[Ric84] Elaine Rich. Natural-language interfaces. *IEEE Computer*, pages 39–47, September 1984.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator.* Springer-Verlag, New York, 1989.

[Sch86] K. J. Schumucker. Macapp: An application framework. *Byte*, 11(8):189–193, 1986.

[Shn83] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, pages 57–69, August 1983.

[SM84] S. L. Smitch and J. N. Mosier. Design guidelines for the user interface to computer-based information systems. EDS-TR-84-190 NTIS No. AD A154 907, USAF Electronics System Divison, Hanscom Air Force Base, MA, 1984.

[Tex87a] Texas Instruments, Austin, Texas. *Explorer Lisp Reference*, 1987.

[Tex87b] Texas Instruments, Austin, Texas. *Explorer Window System Reference*, 1987.

[Tex89] Texas Instruments, Austin, Texas. *Explorer SLE X Window System Programmer's Reference*, 1989.

[WPSK86] Anthony I. Wasserman, Peter A. Pircher, David T. Shewmake, and Martin L. Kersten. Developing interactive information systems with the user software engineering methodology. *IEEE Transactions on Software Engineering*, SE-12(2):326–345, 1986.

[Zei84] Bernard P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation.* Academic Press, London, 1984.

[Zei87] Bernard P. Zeigler. Hierarchical, modular discrete-event models in an object-oriented environment. *Simulation*, 50:219–230, 1987.

[Zei90] Bernard P. Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models.* Academic Press, San Diego, California, 1990.