Definitive Programming:
A Paradigm for Exploratory Programming

by
Simon Yun Pui Yung

A thesis submitted for the degree of
Doctor of Philosophy in Computer Science

Department of Computer Science
University of Warwick
Coventry CV4 7AL
U.K.

October 1992

PAGE NUMBERING AS ORIGINAL

Definitive Programming:
A Paradigm for Exploratory Programming

Simon Yun Pui Yung

Supervisor: Dr W M Beynon

Department of Computer Science
University of Warwick
Coventry, U.K.

A thesis submitted for the degree of Doctor of Philosophy

Abstract

Exploratory software development is a method that applies to the development of programs whose requirement is initially unclear. In such a context, it is only through prototyping and experimenting on the prototypes that the requirement can be fully developed. A good exploratory software development method must have a short development cycle. This thesis describes our attempt to fulfil this demand. We address this issue in the programming language level. A novel programming paradigm – *definitive (definition-based) programming* – is developed.

In definitive programming, a state is represented by a set of definitions (a *definitive script*) and a state transition is represented by a redefinition. By means of a definition, a variable is defined either by an explicit value or by a formula in terms of other variables. Unless this variable is redefined, the relationship between the variables within the definition persists.

To apply this state representation principle, we have developed some *definitive notations* in which the underlying algebras used in formulating definitions are domain-specific. We have also developed an agent-oriented specification language by which we can model state transitions over definitive scripts. The modelling principles of definitive programming rest on a solid foundation in observation and experiment that is essential for exploratory software development.

This thesis describes how we may combine definitive notations and the agent-oriented programming concept to produce software tools that are useful in exploratory software development. In this way, definitive programming can be considered as a paradigm for exploratory programming.

Keywords: definitive programming, programming languages, rapid prototyping, state-transition model, modelling and simulation, software development, agents, human-computer interaction.

Table of Contents

Table of Listings

Table of Figures

Acknowledgements

I wish to express my gratitude to my friends and colleagues who helped me and encouraged me during these years. In particular, I thank Dr. Steve Russ in Warwick and Dr. Rick Thomas in Leicester University for their useful comments. I must thank my brother, Edward Yung for his love, care and practical help while he was with me in the University.

I am most grateful to my supervisor, Dr. Meurig Beynon, for the continuous help, encouragement and constructive criticism he has given me during the work. Especially, I thank him for allowing me to have a year off for developing my personality and communication skills during the course.

I am so much thankful to the Coventry Chinese Christian Church who supported me during my financial hardship so that I may concentrate on my study. And above all, I have to thank God for providing me both the necessary intelligence for the work.

1

Introduction

1.1. Programming Language and Software Development

Programming is more than *translating* what we want the computer to do into a computer program; it involves the whole process of *determining* what basic information the computer needs to possess, *determining* what we want the computer to do, *transforming* the specification into a program and *evaluating* the specification and the implementation. Implementation is only a small step in the software development cycle. Implementation in the JSD software development process, for example, only contributes to one of the six development steps [Jackson83]. What is more, some authors claim that the hard thing about software construction is deciding what one wants to say, not saying it (cf [Brooks86, Sommerville89]).

Although historically programming languages have been concerned with implementation, some kind of programming language is essential as the fundamental communication medium between the participants in the software development process. It is difficult to discount the role of research in high-level languages in solving the

essence of the problem of complex software development [Harel92]. Research on programming language design should focus not only on implementation issues but also on the relationship between a programming language and the whole software development cycle.

Perhaps we can learn a lesson from the development of object-oriented programming (OOP). The idea of OOP (viz programming as object-based modelling) was first brought out by Simula, and can be traced back to the sixties [Naur63, BDMN79]. It was not widely known until the early 80's, when the object-oriented language Smalltalk [GR83] and later C++ [Stroustrup86] were launched. They triggered lots of interest in the programming community. "Suddenly everybody is using it, but with such a range of radically different meanings that no one seems to know exactly what the other is saying", Cox commented [Cox86]. It is difficult to define OOP. Wegner attempted to define it by "object-oriented = objects + classes + inheritance" [Wegner87], but this definition fails to address those object-oriented languages that have no classes. In some object-oriented languages, properties of objects are inherited from other individual objects rather than from classes. Hence, Nelson modifies the definition of OOP to "object-oriented = (objects + classes + inheritance) OR (objects + delegation)" [Nelson90]. Because of the diversity of practice in OOP, Nelson claims that we are creating an object-oriented "Tower of Babel" [Nelson91]. When discussing OOP, many, such as [SB86] and [DT88], refer to techniques like inheritance, message passing and data encapsulation but neglect the object modelling principle. Noticeably from the late 80's, the underlying programming methodology of object-oriented languages is emphasised. For instances, Booch discusses object-oriented *design* (OOD) rather than what an object-oriented *language* can do [Booch91]; Meyer rightly states that the first principle of object-oriented program design is "ask not first what the system does: ask WHAT it does it to!" [Meyer88]. The history of OOP indicates that it is best to understand the application

software design issues in order to give a clear direction to the development of a programming paradigm.

Two important aspects of software design are: it is a sort of design and it is dealing with computation. One difficulty in design is that the software requirement is often unclear or a specification may not be available because the domain of application is poorly understood [Trenouth91]. As Whitefield puts it: "*design is more of a dialectic between the generation of possible solutions and the discovery of the constraints operating on the solution space*" [Whitefield89]. Also Fisher and Boecker state that "*design is best understood as an incremental activity that makes use of existing prototypical solutions to gain a deeper understanding of a problem*" [FB83]. These motivate the exploratory software development process.

One of the earliest proponents of exploratory software development was Sheil. By showing some cases in which any attempt to obtain an exact specification from the client is bound to fail (because the client does not know and cannot anticipate exactly what is required), Sheil concludes that "*no amount of interrogation of the client or paper exercises will answer these questions; one just has to try some designs to see what works*" [Sheil83]. This statement characterises exploratory software development.

One of the examples used in this thesis can illustrate this. During the simulation of a train departure protocol, we discover that it is possible for the train to move while a passenger has opened a door and is attempting to board the train. It is a dangerous act. When the departure protocol is examined, the protocol between the driver, station-master and guard works well in isolation, the passengers also make correct decisions for alighting and boarding. A problem arises only when these two sets of protocol interact. It is therefore difficult to foresee the problem before simulation.

Since there may not be any expected problematic areas in the protocol, it would require a formidable analysis in order to understand the source of the problem before

any remedy can be suggested. In the train example, for instance, a possible remedy might leave the protocol of both the station-master and the passenger unchanged, but add locks to the doors. Because 'understanding' seems to be the bottle-neck of software design, exploratory software development put its emphasis on 'understanding'. Exploratory software development employs a run-understand-debug-edit (RUDE) cycle [Partridge86, PW87]. In this RUDE cycle, experiments are conducted and observations are made in order to understand the behaviour of the software prototype. A main objective in programming paradigm development is therefore to shorten the observation and understanding processes.

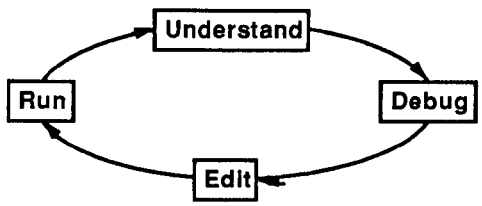


Figure 1.1: The Exploratory Software Development (RUDE) Cycle

There is a common characteristic between experimentation and computation – they are both state-based. Experimentation gives rise to a state-based interpretation of the prototype – what is the state of the prototype when something becomes the input to the experiment? Computation is also state-based. Computation concerns states and the interactions between them. It is natural, therefore, to develop a state-based exploratory programming paradigm.

In addition to being state-based, an exploratory programming paradigm should respect four principles for exploratory software development laid out by Trenouth in [Trenouth91]. Exploratory software must always be: continuously executable, easily extendible, conveniently explorable and usefully explainable. In this thesis, we consider a new approach to programming, and seek to justify the claim that this programming paradigm is suitable for exploratory software development.

1.2. Dependency and Observation

There is a close correspondence between observation and dependency. The reason that we do experiments is because we believe that the input/output relationship of an experiment will be consistent throughout different observations. Therefore, the relationship can be identified or verified through experiments.

The properties we might be interested in changing and observing in an experiment characterise an object. Our approach to software experiment is to capture the dependency information of the properties within an object and between objects by means of definitions. This is why we have called our approach *definitive programming*, meaning *definition-based* programming. A set of definitions, or what we have called a *definitive script*, then records the current state of experiment. In order to change the current state (or to perform an experiment), part of the definitive script is to be redefined. By a definition we mean a formula of the form:

$$x = f(y_1, y_2, \ldots)$$

The value of the variable x is always equal to the evaluation of the formula $f(y_1, y_2, \ldots)$. By defining variables using formulae rather than explicit values, the data dependency of the variables is recorded. The value of x depends upon the values of $y_1, y_2, \ldots$ where $y_1, y_2, \ldots$ may themselves be functionally dependent upon other variables.

A definitive script of this nature is restrictive; it can only capture uni-directional relationships. That is, in a set of definitions, no circular dependency is allowed. On the other hand, this generally guarantees that a set of definitions can be evaluated. Moreover, we believe that the study of 'definitions' will establish a better foundation for more complex relationships such as constraints. This is evident from the fact that some constraint systems, such as ThingLab, Procol and RL/1 [BD86, MBF89, LV91, van Denneheuvel91, CP87], define constraints explicitly or implicitly by sets of methods that can be invoked to satisfy the constraints. Each of these methods serves a

similar function to a definition in our sense. By the appropriate selection of methods, one from each set, the constraints are resolved. This process can be understood as establishing and evaluating a definitive script.

1.3. Motivating Ideas

It is our belief that there is a way of programming that is rooted in modelling dependency between observations. This belief is supported by the following evidence:

1) Research done on definition-based systems

Several definitive notations have been designed and implemented. A definitive notation is a programming notation that can be used for formulating a set of definitions. It is described as a "programming notation" rather than a "programming language" because it only represents part of the information needed for general-purpose programming. DoNaLD and ARCA are two examples of definitive notations. DoNaLD is a definitive notation for 2-D line drawing [BABH86] and ARCA is a definitive notation for displaying and manipulating a class of combinatorial diagrams [Beynon86a]. The data types in DoNaLD and ARCA are application-oriented. For example, DoNaLD has *shape*, *point*, *line* and *circle* whereas ARCA has *diagram*, *colour* and *vertex*.

Definitive systems such as the DoNaLD system ease our understanding and observation of the application in at least two ways. Firstly, the definitive notation is closer to the application than a general-purpose language. The gap in translating between the programming model and the real world is narrowed. Secondly, definitive systems provide immediate feedback. If a box is defined in DoNaLD by lines joining its four corners and the positions of three corners are defined relative to the south-west corner (box/SW), repositioning of the box by redefining the DoNaLD variable box/SW will have an immediate effect on the positions of the four lines on the screen. A short feedback cycle allows a large number of experiments to

be done on the current state in a short time. Later in the thesis, it will be shown that all the qualities of an exploratory programming paradigm – continuous executability, extendibility, explorability and explainability – are present in definitive systems.

In addition to the research in definitive state representation, methods of specifying transitions between states are also investigated. The EDEN definitive language[1] is the contribution of Edward Yung to specifying definitive state transitions in a sequential fashion [Yung89]; [Slade89] provides a thorough study of the LSD specification language for concurrent systems modelling and the ADM programming language for the implementation of LSD. These show that definitive programming is capable of specifying general state-transition models. Hence, definitive programming is an all-purpose programming paradigm which captures data dependency.

2) *Connections between definitive programming and other programming paradigms*

We are actively developing an agent-oriented definitive system. This kind of programming partitions a system into sub-systems according to the agents involved. Every agent has a knowledge of its environment and has its own variables. All these are represented by definitions. An agent will act upon its own understanding of the environment by typically redefining some variables. In this programming paradigm, programming using definitive state representation is similar to functional programming and the agent partitioning is similar to object-oriented decomposition.

---

[1] The term *definitive language* is used to refer to any programming language in which we can formulate definitive scripts.

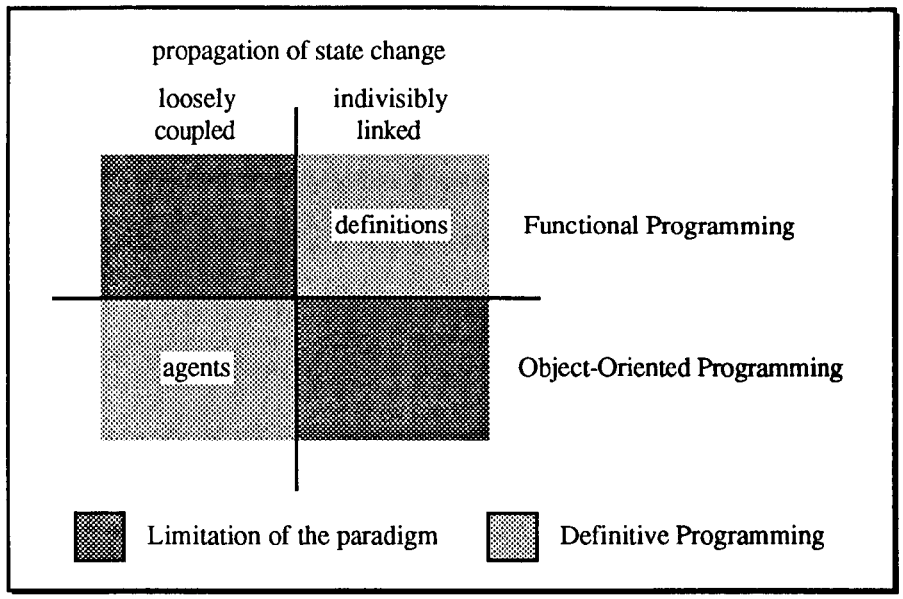


Figure 1.2: Programming Paradigms vs Propagation of State Change

Definitions and agents are associated with two complementary kinds of propagation of state change. Definitive scripts are associated with indivisible propagation (e.g. as in a mechanical linkage) and agents with loosely coupled propagation (e.g. as in asynchronous communication), as illustrated in Figure 1.2.

In a functional program, everything is a function. Writing a functional program is expressing the input/output relationship of an application in functional terms. There is no concept of state in functional programming. This means that the relations expressed in a functional program are of a static nature.

The primary use of functional abstractions in a historical sense is to represent relationships between observations made in the same context. These relationships are associated with modelling indivisible propagation of state change – they correspond to 'definitions' in Figure 1.2. A functional abstraction is not the most appropriate way to model propagation of state change that is loosely coupled, such as commonly arises in interactive programming (see §2.2.2 and §7.1).

Object-oriented programming, in contrast, models propagation of state change in a dynamic fashion through explicit communication between objects. When an

operation performed on one object requires corresponding operations to other objects, this is modelled by means of message passing.

Object-oriented programming simplifies the representation of loosely coupled state changes by reducing the problem of programming a system to that of programming the objects within the system. This is closely connected with the role of agents as in Figure 1.2. Object-oriented programming is less successful in representing indivisible propagation of state change such as is required for synchronisation in concurrent object-oriented models [Baldwin87].

Figure 1.2 indicates that a definitive program which combines scripts of definitions and agent specification exploits the best qualities of the functional and object-oriented paradigms.

3) Broad programming practice

There are few programming paradigms in use in the commercial world but there are many programming paradigms used or under development in research laboratories. Procedural languages dominate commercial computing, but increasing program complexity and improved parallel hardware technology lead us to question their suitability for applications in the future [Turner83, Landin66, Baldwin87]. Many programming paradigms are invented. But will any one of them be *the* future programming paradigm, if there is one?

Baldwin, Hillis and Steele have argued that data parallelism is the key to maximising the utility of parallel hardware [Baldwin87, HS78]. Data parallelism is closely connected with the identification of data dependency. Baldwin compares several programming paradigms with respect to their suitability for multi-processor machines [Baldwin87]. By his analysis, neither conventional procedural programming, object-oriented programming, functional programming nor logic

programming is good for specifying data dependency. In contrast, definitive programming explicitly describes dependency relationships between data.

Baldwin suggests that constraint programming may be the best candidate for parallel programming. However, constraint satisfaction is generally recognised to be a time-consuming exercise. Most of the existing constraint systems either accept only a restricted kind of constraint or use constraint management methods that are given explicitly by the programmer. There are clear connections between definitive scripts and systems of constraints (cf §1.2). Definitive programming may be an appropriate compromise where efficiency, expressive power and convenience are concerned.

In [Smith87], Smith argues that the relation between a program and the outside world should guide the development of new foundations for programming – the traditional account of the semantics of programs is not adequate. Programming paradigm development should also focus on "the semantics of the semantics" of programs. Definitive programming, a paradigm founded on describing the relationship between observations obtained from experiment, may address the essence of the problem [Beynon92, BR92].

The above points indicate that the most widely known and well established programming paradigms have significant limitations. It is entirely possible that a new programming paradigm which is based on modelling dependency would shake the whole programming world.

4) Use of dependency in current systems

Although dependency is rarely formally studied, it is not difficult to identify systems which make heavy use of dependency. The most prominent one is the famous spreadsheet. An electronic spreadsheet is a table of cells in which the relationships between the cells are explicitly written down for the calculation of the

values of the cells. A graphical user interface tool is another example. A stylesheet in word processors is yet another. Spreadsheets and style-based word processors may be the most commonly used software tools, and most probably the explicit use of dependency is a crucial reason for their success.

1.4. What Has to be Done

Our research in definitive programming can be logically divided into two sub-areas: representation of states and modelling of transitions. In the area of representation of states, definitive notations are designed and implemented to evaluate and explore the advantages and disadvantages of definitions. In the area of modelling transitions, higher-level specification and control languages are designed to govern the transitions of states. Furthermore, a theoretical framework needs to be developed. Practical examples are also required in order to evaluate the research in every stage.

1.4.1. Brief History of Definitive Programming Research

It would be helpful to present a short history of the research in definitive programming to give some flavour of the scope the research involved. Whilst the definitive paradigm has a relatively short history, definitive principles have been used informally to maintain relationships between values of variables for a very long time. Early examples include the specification of machining sequences in numerical machine tools in APT [ITT67], Wyvill's interactive graphics language [Wyvill75] and the electronic spreadsheets of the early 70's. The first paper to describe the abstract concept of a definitive notation was [Beynon85], published in 1985. Independent definitive notations and an agent privilege specification language were then developed in parallel.

In the area of developing definitive notations, there were only two definitive notations designed before 1987. They were ARCA and DoNaLD. Amongst them only ARCA was implemented. One reason for the slow development of definitive notations was that implementing a definitive notation was a time-consuming job. Since the

design and implementation of the definitive language EDEN in 1987, the implementation job of definitive notations is greatly eased. The name EDEN is, in fact, an abbreviation of "an Engine for DEfinitive Notations". The implementation of a definitive notation becomes a task of producing a translator from the specific notation to EDEN and writing an associated EDEN library for the simulation of the data types and underlying algebra in the notation. This is a much simpler job than writing the evaluation engine for a new family of definitions. However, definitive notations were still running in isolated environments. Within a session, one could only interact with a single definitive notation other than EDEN.

Much of the development of an agent privilege specification language was carried out by Mike Slade. The LSD notation, first defined in 1986 [Beynon86b] and subsequently modified in 1989 [Slade89], was a result of this research. LSD is a specification language for the behaviour of multi-agent reactive systems. An LSD specification is not executable; it has to be transformed manually into the ADM language before execution [BSY88]. The ADM definitive language was designed both for the interpretation of LSD and to give a more satisfactory abstract account of the hybrid programming paradigm used in EDEN, and subsequently implemented for the former role by Slade. The main problem with ADM is its limited data types. This restricts the usefulness of ADM and hence hinders the development of LSD.

1.4.2. The Future

Ideally, the ultimate system will be very large. In that system, many definitive notations will describe different parts of a program. Possibly these definitive notations will be defined within a more powerful and general language. The LSD specification language has plenty of scope for improvement as well. Ideally, we should like to be able to specify the methods of conflict resolution for agent actions in LSD. Also, the transformation from LSD to an executable program should be simpler. A possible solution is the use of hidden text to annotate an LSD specification. By the addition of

simulation decisions in this way, the annotated LSD specification will be executable, in principle. The ability to specify conflict resolution also implies the ability to model higher level data dependencies, such as constraints.

Although we have some hints of what this ultimate system will be like, it still seems to be a long way before a preliminary version can be prototyped. More research on the LSD notation itself, the transformation process and the linkage between definitive notations has to be done beforehand.

1.5. Contribution of This Thesis

This thesis is not intended to overcome all the obstacles to the ultimate system. Its main objective is to merge previous research efforts in definitive programming around a unified theme. It becomes apparent that 'Programming as Modelling' is one of the major contributions of definitive programming to the software development process [BRY90, BBY92, BY92], and it is in this context that previous researches merge. Definitive notations, by having their specific domains and being definitive in nature, are suitable for modelling states of the real world, while the agent privileges described by LSD are suitable for modelling the dynamic behaviour of the real world. Because of its strong modelling orientation, definitive programming satisfies the requirements for exploratory programming – it is state-based, continuously executable, easily extendible, conveniently explorable and usefully explainable. Apart from abstract discussion of the potential of definitive programming, my practical contribution is to combine several definitive notations, and to some extent LSD, into a single programming environment based on definitions. This brings us practically a step forward to our vision of programming.

This thesis will describe both practical work done and the philosophical advancement in definitive programming. The areas covered are:

Practical work

1. *In the area of definitive representation of state:* A definitive system, Scout, in which several definitive notations can be used cohesively, is designed and implemented. This involves modifications to the existing implementation of definitive notations as well as developing a general interface program to the window system.

2. *In the area of transition of states:* Firstly, the source of definitions of the system is widened. Originally, the only way of introducing new definitions was via textual input; now definitions can be generated by mouse events as well as generated by the system itself. Additional sources of input allow the system to respond to richer form of interaction with its environment. Secondly, an ADM-to-EDEN translator is prototyped. Since ADM is an implementation language for LSD and EDEN is the underlying language of the Scout system, the ADM-to-EDEN practically links the previous research works together. This work also indicates how the practical power of EDEN can, in principle, be expressed in the purer programming paradigm of the ADM where *all changes* of state are represented by redefinitions.

3. *In the understanding of definitive notations:* The definitive notation Admira is prototyped. Since the evaluation mechanism of Admira makes use of the functional programming system Miranda [Turner86], studying Admira is a means to understand the relationship between definitive programming and functional programming.

Philosophical advancement

This thesis:

1. links up much previous work in the entire definitive research programme. Sets of definitions to define state and agent-oriented programs to describe transition both have a strong modelling foundation. This serves as the link.

2. develops the idea of using definitions for modelling the real world. By means of definitions, the gap between a computer programming model and the real world is narrowed.

3. advocates the new concept that definitive programming is good for exploratory software development.

4. evaluates, by means of illustrative examples, the advantages and limitations of current definitive system. In the course of this discussion, it will be demonstrated that definitive notations can play a significant part in general-purpose definitive programming.

1.6. Outline of the Thesis

This thesis is organised so as to defend the claim that definitive programming is a programming paradigm that is suitable for exploratory software development. In the next chapter, the heart of definitive programming – the Definition-based State-Transition (DST) model – is introduced. The abstract virtues of the DST model will also be explained. Chapter 3 shows that some commonly used software tools are already using concepts and techniques close to our notion of definitions. Chapter 4 describes my design and implementation of the Scout definitive notation. Scout is a definitive notation for describing screen layout. By means of an illustrative example, Chapter 5 demonstrates how the Scout notation assists exploratory screen layout design. Chapter 6 describes my work on integrating several definitive notations. Through integration of definitive notations, we can broaden the domain for our exploration. Chapter 7 stands between the discussion of definitive representation of state and specification of transitions over such a representation. It discusses the ways in which the power of single-agent definitive systems, such as Scout, may be enhanced. The discussion prompts us to introduce more general agents into definitive systems. Chapter 8 describes an agent-oriented specification language (LSD). By

describing a software tool for assisting the implementation of LSD and giving practical suggestions for improving LSD, this chapter advocates that agent-oriented definitive programming can not only deal with all-purpose programming but is also suitable for exploratory development of software. Chapter 9 summarises the thesis and concludes that definitive programming is a good paradigm for exploratory programming.

2

Definition-Based State-Transition Models in the Abstract

It has been mentioned in the introductory chapter that our system may ultimately comprise many notations. Our current system can already relate six notations: DoNaLD, ARCA, Scout, EDEN, ADM and LSD. DoNaLD is a definitive notation for 2-D line drawings [BABH86]; ARCA is a definitive notation for displaying and manipulating a class of combinatorial diagrams [Beynon86a]; Scout is a definitive notation for describing screen layout [Yung88]; EDEN is a general definitive language for arithmetics, strings and lists [Yung87, Yung89]; ADM is a parallel definitive language [Beynon88b, BSY88, Slade89] and LSD is an agent protocol specification language [Beynon86b, Slade89]. Each of these notations addresses a specific domain. For this reason, each has its own set of data types and syntax. With such diversity of

notations, it is easy to lose focus on what the essence of definitive programming is. Therefore, this chapter will abstractly describe the core of the definitive paradigm – the Definition-based State-Transition (DST) model – before we discuss particular aspects of these definitive notations or languages in more detail in later chapters. This chapter will also discuss the virtues of definitive programming in relation to exploratory software development.

2.1. The Definition-Based State-Transition (DST) Model

In the early stages of the research on the definitive paradigm, emphasis was laid on generalising the "spreadsheet" principle to more general programming notations [Beynon85, Beynon88a, Beynon89]. Until the development of the LSD notation in 1986, the kind of interaction involved was still confined to "redefinitions by user". The LSD notation described a system in terms of processes (and later agents [Slade89]) interacting with each other. Since then, research on definitive programming has been widened to address general-purpose programming. Of particular interest in our research is the programming principle embedded in what we have called *definitive programming* [BNS88, Beynon88b, BRSYY89, BNRSYY89, BSY90], viz the use of sets of definitions to represent computational states. The understanding of definitive programming in terms of states and transitions evolved gradually; the phrase "definition-based state-transition model" first appeared in our papers as recently as 1989. The major work on studying the DST model started then.

A definition-based state-transition (DST) model is a state-transition model in which a state is represented by a set of definitions – a *definitive script* – and a transition is represented by a redefinition. A redefinition has essentially the same significance as a definition. The term *redefinition* is appropriate because a definition will overwrite the previous definition of the variable concerned whilst, if the variable has no current definition, the new definition will be added.

If a definition has to be discarded from a state, it is equivalent to redefining the variable by a special undefined value. This is because we can imagine that the state is a universal set of variables in which the variables are defined with undefined values by default.

2.2. Comparison of the Concept of State and Transition in Different Programming Paradigms

Petre and Winder categorise programming languages along a continuum between two extremes of computational models – the imperative model and the declarative model [PW88]. The imperative model is the computational model of the von Neumann machine. This is a model of "computation by effect". Under this model, algorithms are expressed as a sequence of changes of states. An imperative program contains explicit instructions for controlling the flow of execution. The declarative model, on the other hand, is a model of "computation by value". There is no sense of instruction in a declarative model, instead there is a "script"[1] which defines what is to be computed. Petre and Winder argue that there is a *continuum* of languages associated with the shift from an imperative to a declarative style that involves the transference first of explicit control and then of algorithmic information from the program description to the implementation.

It is obvious that the imperative languages are state-based languages. The declarative languages are arguably stateless in the sense that they describe an abstract input/output relationship rather than any computational state. From another perspective, we might reason that a script is concerned with the description of only one

---

[1] David Turner introduced the term "script" for the programs written in his functional languages to emphasise that such programs were qualitatively different from their imperative counterparts [Turner85].

state – the encapsulated behaviour of the required program (cf Chapter 7). In either way, the concept of state is not significant in declarative languages.

It is generally accepted that procedural programs are hard to verify and are more difficult to adapt onto parallel machines [Baldwin87]. However, we believe that some kinds of activities in the real world are most conveniently described by procedures. For example, a person may like to pick up a book from a bookshelf, walk to a desk and put the book on the desk. The behaviour of a person naturally comprises a sequence of actions which is most appropriately represented by a procedure.

In definitive programming, a state is prescribed by a definitive script. The maintenance of values of variables in a definitive script is similar in spirit to declarative programming in that it involves implicit evaluation of expressions. In contrast to declarative programming, definitive programming does not presume a declarative style of specifying transitions. (Indeed, later in the thesis, we advocate the use of an agent-oriented style for specifying state transitions.) There is, therefore, scope in definitive programming to explore the virtues of both declarative and procedural programming.

Applying Petre and Winder's classification method, the position of the DST model in the continuum from "imperative" to "declarative" is somewhere in between the two extreme models but its bias may vary depending upon the way of specifying state transitions. On one hand, the DST model has variables and a concept of state. In this way, definitive programming is similar to the imperative model. On the other hand, the values of the variables are not necessarily directly modified by a program instruction (there may not be any). This is because a definitive variable is fundamentally defined by a formula instead of an explicit value. That is, its value is determined by what it is asserted to be rather than by direct assignment. Therefore, if the language that governs the state transitions is procedural, the DST model would be biased towards the imperative model. Otherwise, the DST model would be biased towards the declarative model.

In view of our freedom to choose the specification style for state transitions, we can identify the main characteristic in the DST model to be its novel approach to state representation. Therefore, in the rest of this section, we will focus on comparing the concept of state in different programming paradigms.

A definitive script specifies the following information pertaining to a state:

1) a collection of values (values of variables),
2) references to the components of the state (variable names),
3) data dependency information between components of the state,
4) methods of maintaining the state (formulae).

In the following sub-sections, the state representation methods of conventional imperative programming, functional programming and object-oriented programming are compared with that of definitive programming.

2.2.1. Conventional Imperative Programming

In conventional imperative programming, a state is a collection of variables containing explicit values. The machine changes the state by assigning new values to the variables. The connection between the values of the variables cannot be observed by looking at one state only. The meaning of a variable cannot be understood without referring to the program; the meaning of a variable may even change from one state to the other during program execution. For example, in the following program fragment

```
1  sum := 0;
2  for I := 1 to N do
3       sum := sum + a[I];
4  mean := sum / N;
5
6  sum := 0;
7  for I := 1 to N do
8       sum := sum + (a[I] - mean) * (a[I] - mean);
9  sum := sum / N;
```

the meaning of sum in line 4 is the summation of N numbers but the meaning of sum has changed to the variance of the N numbers after line 9, and at other points, sum is a storage of intermediate results.

Definitive programming, to certain extent, gives meaning to the variables. A definitive variable is defined by a formula. This formula is the 'value' of the variable. The formula prevails until the variable is redefined by another formula. However, the value of the formula may change over time as the variables in the formula are redefined. Therefore, there are two levels of understanding a definitive variable: knowing its interpretation (associated with the formula itself) and knowing its current value (the value calculated from the formula).

Backus in his much referenced paper "Can Programming be Liberated from the von Neumann Style?" [Backus78] points out two main problems with conventional languages: word-at-a-time bottleneck and splitting programming into an orderly world of expressions and a disorderly world of statements.

Word-at-a-time bottleneck is the input/output limitation of the von Neumann machine model. This is reflected by the basic operation in a conventional procedural language – each atomic state transition allows a change to the value of just one variable (a single assignment). Because of advances in computer architecture, this word-at-a-time bottleneck no longer applies to computer hardware. It is now the conventional procedural language that imposes this bottleneck. Definitive programming breaks this bottleneck by allowing indivisible changes of values of many variables in a single transition of state. When the definition of a variable is changed, not only the value of this variable will be updated but also the values of those variables defined in terms of this variable.

Backus uses the phrase "the orderly world of expressions" to refer to the expressions on the right hand side of assignment statements. He claims that an

expression has useful algebraic properties whilst a statement has few useful mathematical properties. Using expressions in the context of assignments destroys the usefulness of the algebraic properties of expressions by side effects. In contrast, the definitive state representation preserves the usefulness of the algebraic properties of expressions by persistently associating the expressions with variables. A change of value induced by the change of other variables does not alter the expression associated with that variable.

2.2.2. Functional Programming

Functional programming and definitive programming are, in principle, not comparable concerning states and transitions because functional programming is stateless. A functional script defines what is to be computed rather than how to compute the target value. Neither is there a concept of procedural variable in functional programming; a mathematical variable is officially not allowed to vary [BR89].

A disadvantage of functional programming arises also from the lack of the concept of state; such a concept is almost indispensable for describing states and transition in interactive programs. Dataflow languages (a branch of functional languages) are more promising in handling interactive programming. Wadge's VISCID program is an attempt to write a vi-like[2] screen editor in LUCID [Wadge85]. Other attempts at writing screen editors in other non-procedural languages like Prolog and Lisp[3] used side-effects and imperative features; Wadge tried to show using VISCID that it is in fact possible to write non-trivial and non-mathematical applications within the constraints of a functional language. However, VISCID relies on the lazy evaluation strategy to control what Wadge has called "internal memory" variables. (Lazy

---

[2] Vi is a standard UNIX full screen text editor.

[3] Because there are imperative features in Lisp, it is seldom considered as a functional language. However, we can extract a functional subset from Lisp [GHT84].

evaluation (call-by-need) is used in preference to eager evaluation, where a function is evaluated as soon as all the arguments are evaluated. If LUCID ran using an eager evaluation strategy, VISCID could only perform batch mode editing rather than interactive editing.) Based on the fact that the execution of VISCID relies on a particular evaluation strategy and that it has a notion of internal memory, we shall argue that although we *can* use a functional language to program interactive applications, programming in a state-based language would be a more satisfactory solution.

Although definitive programming and functional programming adopt significantly different programming models, a definitive script (a set of definitions) and a functional script have a useful mathematical property in common: a script (in either paradigm) will always evaluate to a unique set of values. It is even plausible to argue that a functional script is a definitive script (see §7.1).

Hudak did a survey on functional programming [Hudak89]. The survey includes a discussion on the active research areas in functional programming. It is interesting to note that there are researches going on in the direction of integrating functional and imperative programming [Lucassen87]. It has been indicated in the last sub-section that we are not promoting imperative programming. However, we emphasis the importance of state-based programming. Although definitive scripts and functional scripts are superficially similar, their interpretation is fundamentally different: our systems recognise on-line redefinition of scripts as part of the computation. This provides a basis for interactive programming using definitive representations of states.

2.2.3. Object-Oriented Programming (OOP)

Many of the ideas behind object-oriented programming have roots going back to SIMULA [DN66]. The first substantial interactive, display-based implementation was the SMALLTALK language [GR83]. Associated with the widespread use of the C language, extensions of C – such as Objective C [Cox84, Cox86] and C++ [Stroustrup84, Stroustrup86] – are also widely used. There is one thing in common

with all these languages – they are all procedural. Hence, OOP gives some people a first impression that it is fundamentally procedural. However, there are a considerable number of object-oriented extensions to non-procedural languages. Loops [BS81], CommonLoops [BKKMSZ86], OakLisp [LP86] and CommonObjects [Snyder87] are some object-oriented extensions to Lisp; SCOOP claimed to be object-oriented Prolog [VLM88]. Hence, it is possible to merge OOP with other programming paradigms. Object-oriented programming is more appropriately understood as a design philosophy [Meyer88, Booch91, WP88].

Cox has described object-oriented programming as an evolutionary development from procedural programming [Cox86]. In particular, the concept of data in object-oriented programming has evolved from a procedural framework. An object is more than a simple value (for example a floating point number in Fortran), or a group of values (for example a structure in Pascal); an object has some methods associated with it to specify how it is to be maintained.

Definitive programming can be viewed as a different kind of evolution from data specification in a procedural style. Definitive programming enriches the meaning of data by assigning to the variables formulae instead of plain values. In a way, we may consider that a definition has already provided a method (the formula) for the maintenance of the variable defined, so that simply grouping the related variables together has a flavour of object-oriented programming. This suggests that object-oriented programming and definitive programming can be usefully combined. Yung has given some suggestions for object-oriented EDEN [Yung89] and Chapter 7.2.3 in this thesis includes a proposal for adding inheritance to the DoNaLD notation.

2.3. Virtues of the DST Model

2.3.1. Data Dependency, Concurrency and Consistency

An important area of concern in studying different programming paradigms is the support for recording and retrieving data dependency information. Data dependency information is useful in two areas: concurrent programming and program development.

With data dependency information, the compiler can automatically distinguish when two operations must be done sequentially because one produces or destroys a value that the other needs. Therefore, the more easily the data dependency information can be obtained, the better the program is suited for parallel processing.

In definitive programming, an acyclic graph of dependency can be drawn from a set of definitions and concurrent updating of the variables can be performed in each layer of the graph. Such a scheme for concurrent maintenance of definitions is discussed in depth in [Yung89]. This way of parallelisation is a kind of data parallelisation (i.e. parallel evaluation of data) which can be determined implicitly by the system. Since it is hard to prove the correctness of those parallelisation schemes given explicitly in a program, implicit parallelisation is more reliable. Moreover, data parallelism is highly effective [Baldwin87, HS78]. Definitive state representations seem to be a good foundation for concurrent programming.

Turner [Turner83] and Landin [Landin66] predicted the future trend for the development of programming languages would be non-procedural languages. One reservation they have about the development of non-procedural languages is that "on conventional von Neumann computers, non-procedural language runs two to three orders of magnitude slower than traditional imperative languages" [Turner83]. Turner suggested that non-procedural languages can be used as effective tools for software prototyping while waiting for the development of systems suitable for non-procedural languages. In fact, many non-von Neumann system architectures are developing: for

example dataflow machine architecture [Veen86, Sowa87], dataflow / von Neumann hybrid architecture [Iannucci88], parallel logic inference machine [Clocksin87, Jorrand87] and object-oriented computer architecture [Harland88].

Speed of execution is significant, but the efficiency of developing a program is of equal importance. The identification of data dependency provides useful information in maintaining a program during program development. Consider the following scenario. Suppose that the value of a in an imperative language, at a stage of program development, had to be maintained to the same value as $2 \times b$, but some time later the programmer determined to alter this assertion to "a equals to $3 \times c$". Then what the programmer has to do is to remove all the assignment statements of the form "$a = 2 \times b$" and insert assignments "$a = 3 \times c$" after each modification of the variable c. If some of the "$a = 3 \times c$" statements were, by mistake, not inserted, the old value of c would be retained in a at certain points of the program. Or if some of the "$a = 2 \times b$" statements were not deleted, then a might obtain a value totally unrelated to c. Understanding data dependency assists the programmer to know exactly what actions have to be done to the program when the specification is altered.

Definitive programming not only makes use of the data dependency information to maintain the consistency of data, it actually prevents inconsistency. Because there is only one persistent definition of a variable stored inside a state, no redundant information and hence no potential inconsistency of data will be found in a definitive script[4].

---

[4] Relational database theory also acknowledges that redundancy leads to potential inconsistency. The relational database designer prevents update anomalies (potential inconsistency) by decomposing a large database into normal form [Ullman82]. Basically, the decomposition schemes that are employed group the fields of the database into sub-databases according to the dependency of the fields. A definition is similar to a single relation within a relational scheme in that only related variables are linked together.

2.3.2. Support for Incomplete Models

It is clear that, in certain contexts, such as during the construction or modification of a model, some variables are not evaluable because the dependent variables are not defined. This does not affect other parts of the model that do not depend on these undefined variables. The partially completed model may still have a meaning: a room without furniture is still a room. Even when a definition depends on undefined variables, its defining formula is also meaningful, not least for the purposes of analysis. A definition limits the possible values of the variable. This point will be elaborated in the next section.

2.3.3. Possible Transformation

Just looking at a set of values gives us little information for reasoning about what these values mean and how they should be manipulated. The following two DoNaLD specifications[5] both produce the shape shown in Figure 2.1.

By looking at the shape alone, it is not possible to guess which specification is the one that generates this shape. This shape may represent a file cabinet with its drawer opened, or it may represent a LED display which is showing the digit 8. A correct interpretation of the shape can only be made with reference to the underlying model in mind, which means a state of an object is more than a set of values (say pixel values). Definitions relate the state and the model in such a way that changes in the model reflect changes of external state; the possible transformations to the object are described in a set of definitions.

---

[5] The example is taken from [BCRY90].

```
openshape cabinet
within cabinet {
  int     width, length
  point   NW, NE, SW, SE
  line    N, S, E, W

  N = [NW, NE]
  S = [SW, SE]
  E = [NE, SE]
  W = [NW, SW]

  width, length = 300, 300

  SW = {100, 200}
  SE = SW + {width, 0}
  NW = SW + {0, length}
  NE = NW + {width, 0}

  openshape drawer
  within drawer {
    boolean  open
    int      length
    line     N, S, E, W

    length = if open then ~/length else 0
    open = true

    N = [~/NW + {0, length},
         ~/NE + {0, length}]
    S = [~/NW, ~/NE]
    W = [~/NW + {0, length}, ~/NW]
    E = [~/NE + {0, length} , ~/NE]
  }
}
```

```
openshape led
within led {
  int      digit
  point    p1, p2, p3, p4, p5, p6
  line     L1, L2, L3, L4, L5, L6, L7
  boolean  on1, on2, on3, on4, on5, on6, on7

  digit = 8

  p1 = {100, 800}
  p2 = {100, 500}
  p3 = {100, 200}
  p4 = {400, 800}
  p5 = {400, 500}
  p6 = {400, 200}

  on1 = digit != 1 and digit != 4
  on2 = digit != 0 and digit != 1 and digit != 7
  on3 = digit != 1 and digit != 4 and digit != 7
  on4 = (digit == 0 or digit >= 4)
        and digit != 7
  on5 = digit == 0 or digit == 2 or digit == 6
        or digit == 8
  on6 = digit != 5 and digit != 6
  on7 = digit != 2

  L1 = if on1 then [p1, p4] else [p1, p1]
  L2 = if on2 then [p2, p5] else [p2, p2]
  L3 = if on3 then [p3, p6] else [p3, p3]
  L4 = if on4 then [p1, p2] else [p1, p1]
  L5 = if on5 then [p2, p3] else [p2, p2]
  L6 = if on6 then [p4, p5] else [p4, p4]
  L7 = if on7 then [p5, p6] else [p5, p5]
}
```

Listing 2.1: Two DoNaLD Specification for Describing the ⊟ Shape

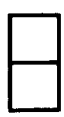


Figure 2.1: An ⊟ Shape

If we are interested in the ⊟ shape only, so that no more change to the shape is needed, this information about possible transformation becomes redundant. Confusion may arise here as to whether the transformation information should be classified as part of a state. The answer can be established using the following illustration. In a

procedural graphics drawing package, it is possible to transform a geometric object to another geometric object via addition and deletion of line segments or other operations. A transformation from a  shape to an  shape may take the following sequence: , and . Does it mean "3 + two lines = 8"? Of course not. The interpretation of this shape as the number 8 cannot be justified. Using definitions to represent a state of an object models the object more faithfully because, on top of a set of values, the possible transformations about the object are described as well.

2.3.4. Exploratory Software Development

To assist in exploratory software development, definitive programming provides:

- a modelling principle: a set of definitions *models* a state. This helps in the understanding phase of software development.
- data consistency. By means of definitions, values of variables will be maintained to their associated formulae. This implies fewer errors during editing a definitive program and easier for debugging.
- good prospects for efficient execution. The potential for data parallelism in evaluating definitive scripts is an advantage for allowing more explorations in shorter time.

These lay a solid foundation for developing definitive programming into an exploratory programming paradigm.

The possibility for software exploration makes reasoning about the properties of definitive programs difficult. This reasoning issue has to be addressed in the future, it is probably associated with the issue of specifying the intended use of definitive programs and the privileges of the users.

3

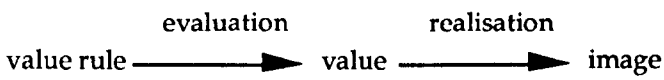
Software Tools Using Dependency

Definition-based (definitive) programming sounds like a new subject, but many software tools, even some that are very popular, use concepts similar to definitions. These software tools include spreadsheets, some document processing software, some graphics editors and the make utility. Such tools can be seen to represent good areas of application for the Definitive State-Transition (DST) model. Their success can also encourage us to pursue definitive programming. This chapter examines how they use dependency information and compares their approach with our use of definitive principles.

3.1. Spreadsheets

3.1.1. Basic Concepts of a Spreadsheet

The functionality of the spreadsheet has increased tremendously since the first spreadsheet program VisiCalc in the 1970's. It can now be used as a database. It can also produce colourful graphs and so is becoming a presentation tool. However, the basic concepts of spreadsheets have not changed. A spreadsheet is basically a collection of cells located on a rectangular grid. Each cell may be referenced by its position on the grid. Each cell may store a formula returning a real value or a string of characters. This formula may or may not contain references to other cells. If it does, then when those cells referenced change value, the formula will be recalculated automatically. The value obtained will then be displayed according to a format rule (often chosen by menu selection). For example in a financial setting, a value often represents the currency and so is appropriately displayed as a number with two decimal places and preceded by a pound sign. In short, the image that appears in a cell has gone through the following process:



In many ways a spreadsheet program is similar to a script of definitions:

- Each definitive variable is analogous to a cell in a spreadsheet.
- A definitive variable may be defined in terms of explicit values or by a formula (value rule) in terms of other definitive variables.
- One of the definitive notations, Scout, is intended to perform a similar role to the spreadsheet format rules (see Chapter 4 and 5).
- Both definitions and spreadsheet programs define uni-directional data dependency.

Although a spreadsheet is very similar to a definitive script, we can see the essential differences between spreadsheet programming and definitive programming in the following three aspects: the variable names, the range functions and operations, and the order of evaluation.

3.1.1.1. Variable names

Superficially, the variable naming system in a spreadsheet is very simple – the name of the cell is the position of the cell on the spreadsheet. The convention is that the name of a column is a group of letters, A-Z, AA-AZ, BA-BZ etc. Column A means the first column, column B is the second, column AA is the twenty-seventh, column AB is the twenty-eighth and so on. The name of a row is an integer starting 1 or 0 depending on the spreadsheet. On deeper analysis, the variable naming system is much more complex.

We can insert a row of cells at any position of a spreadsheet. By doing so, other cells at or below the insertion point will be moved down one row. This means that the cells at or below the insertion point will obtain new names. Similar situations are the insertion of a column, deletion of a row or a column. Therefore, the first observation is that the name of a cell in a spreadsheet may change over time.

An implication of changing a variable's name (cell name) is that the formulae in other cells may require corresponding changes. Suppose that a series of cells C2 to C5 (denoted by C2..C5) are intended to show the multiples of C1. The definitions are:

Cell	Formula
C1	3
C2	C1 + C1
C3	C2 + C1
C4	C3 + C1
C5	C4 + C1

The insertion of a row should, and in a spreadsheet typically does, redefine the definitions of the cells to the following:

Cell	Formula
C2	3
C3	C2 + C2
C4	C3 + C2
C5	C4 + C2
C6	C5 + C2

The change of formula is, however, not always desirable. For instance, if one moves cells C2..C5 to D1..D4, one may like to leave references to C1 untouched. That is:

Cell	Formula
C1	3
D1	C1 + C1
D2	D1 + C1
D3	D2 + C1
D4	D3 + C1

This can be achieved by other conventions of the naming scheme. If a $ sign is put before the coordinate of the cell, that coordinate will not be subject to change. For example:

Formula of cell C2 before move	Formula of cell D1 after move
C1 + C1	C1 + C1
$C1 + $C1	$C0 + $C0[1]
C$1 + C$1	D$1 + D$1
C1 + C1	D0 + D0

[1] If the row number starts from 1, the formula will be "=$C1 + $C1" instead.

The flexibility in the referencing system in a spreadsheet is an advantage of the tabular arrangement of cells. The new reference of a cell can be calculated simply by adding the offset of the cell displacement to the original reference.

3.1.1.2. Range Functions and Operations

The spreadsheet also takes advantage of the regularity of the variable names in providing a rich set of range functions and commands. A range in a spreadsheet is a collection of cells enclosed by a rectangle defined by the upper left and lower right cells in the region. Functions such as @sum (summation of all the cells in the range), @stddev (standard deviation) and commands such as fill a range with incremental values are usefully defined for ranges.

The benefit of having ranges is that a variable number of cells may be addressed together. If a cell is inserted in a range whose summation is performed elsewhere, there is no need to alter the formula for the summation. The range of the summation is automatically extended by the re-adjustment process of reference names described earlier. Without range functions, a summation would require an additional term in the expression.

3.1.1.3. Order of Evaluation

Is the order of evaluation important in a spreadsheet? Order of evaluation is not important so long as the formulae contain no circular referencing. Unfortunately, a spreadsheet normally allows circular referencing. As a result, spreadsheets need commands to specify the recalculation order (row order or column order) and the stopping condition. The stopping condition is either when the evaluation result stabilises or the user-specified maximum number of iterations is reached.

3.1.1.4. Differences Between a Spreadsheet And a Definitive Script

The first difference between a spreadsheet and a definitive script is in the interpretation of formulae. The fact that spreadsheets allow circular referencing indicates that spreadsheet programming tends to interpret the value rules as methods of maintaining the values of the cells rather than stress the 'real-world semantics' of the cells. In effect, a value rule in a spreadsheet may serve as a procedural computational device. Definitive programming in contrast tries to maintain that formulae are statements about the relationship between definitive variables.

The second difference is that a definitive script describes the relationships between variables whilst a spreadsheet basically describes relationships between the values on particular locations or relative locations on the spreadsheet. Depending on the way the 'variable names' within a formula are defined, the formula relates the current cell with cells on particular locations or relative locations of the spreadsheet, not fixed cells. This could be a reason for classifying the spreadsheet as a visual programming language [Myers89]. On the other hand, a definitive variable is similar to a conventional variable in that the variable name always refers to the same entity. Because of the geometric referencing characteristic of spreadsheets, spreadsheet programming can take advantage of the tabular arrangement of cells. Range functions can be easily implemented in a spreadsheet but are not that trivial in definitive programming. However, a definitive variable is usually named after what it is meant to represent in the real world. A variable name has a message to tell which is often more important than the location of its visual representation.

3.1.2. Appraisal of Spreadsheet Programming and Definitive Programming

Whilst many people today are still treating the spreadsheet as a user-friendly interface, some like Kay and Kokol do treat the spreadsheet as a programming paradigm. Not

only that, they consider that the spreadsheet is in fact an ultra-high level language [Kay84, Kokol88]. Some like Hewett and Green also suggest that many features of electronic spreadsheet are helpful in rapid prototyping and notation design [Hewett89, Green89]. We find that many of their arguments are also applicable to definitive programming.

3.1.2.1. What-If

The usage of spreadsheets has been expanded over the years. A frequent use of spreadsheets is in modelling and simulation, which has been applied to engineering, chemistry, neural network, ecology, physics and psychology [Hewett89]. The *what-if* feature of the spreadsheet naturally makes it a tool for forecasting when some condition will pertain in the future. Typical examples are financial modelling and sensitivity analysis which takes advantage of this 'what-if' ability [Jackson88].

In order to adapt to the application, it is sometimes necessary to extend the underlying algebra of a spreadsheet. DYNAGRAPH is a spreadsheet-based interactive simulation modelling system. There are functions in DYNAGRAPH, such as table look-up functions, forecasting functions and delay functions, that are particularly designed for modelling and simulation of multi-period planning. "It would be wearisome, if not impossible, to use the popular spreadsheets *for the job*" [Anonum88].

Since definitive programming also maintains that a variable will be updated whenever one or more of the variables on which it depends is updated, *what-if* is also a prominent feature in definitive programming. Modelling and simulation is also a major application area of definitive programming. In fact, many of our papers discussed with examples the application of definitive programming in modelling and simulation [BBY92, BY92, BSY90, BRY90, BNS88].

Definitive programming faces a similar issue to the spreadsheet – the need to extend the underlying algebra for particular kinds of modelling and simulation. From the beginning, definitive programming has involved special-purpose *definitive notations* [Beynon85]. These definitive notations have underlying algebras specially designed for certain applications.

3.1.2.2. Rapid Prototyping

Hewett [Hewett89] suggested some considerations for developing rapid prototyping environments. The environment provided by a spreadsheet addresses these considerations:

(A1) Eliminate or reduce the need for both developer and user to attend to I/O details during prototype developments.

(A2) Allow the developer flexibility in creating alternative user views of the prototype.

(A3) Make it easy for the developer to change underlying relationships and parameter values, and to introduce simplifying assumptions.

(A4) Require limited programming from the developer during the process of prototype development.

(A5) Make possible easy replication of differing versions of the interface for comparative examination and testing by both developer and user.

(A6) Allow the developer to support the user's intuitive understanding of the task though direct representation of significant features of the task environment in which the system will be used.

(A7) Allow for the development of separate interface modules and layers, and links among those functional units.

(A8) Offer the pedagogical value of being accessible to, learnable by and modifiable by others, including users, under some circumstance.

Hewett also points out some limitations of spreadsheet regarding rapid prototyping:

(L1) a limited number of cells,

(L2) lack of support for building up and modifying new interface primitives,

(L3) lack of support for the development of a set of higher level design abstractions,

(L4) little or no provision for tracking the history of the design process,

(L5) lack of debugging facilities,

(L6) no means to constrain the end-users' interaction.

Definitive programming has much in common with spreadsheets concerning the advantages listed above. Empirical evidence from the use of our software prototypes by the undergraduate project students indicates that, apart from advantage (A5), which is unique to the spreadsheet and derives from its convenient copying and moving facilities, the advantages of spreadsheets cited by Hewett are shared by definitive programming. Since definitive notations provide a richer set of data types, operators and visual representations, some advantages such as flexibility in creating alternative user views and pedagogical value are further enhanced by definitive programming.

At the same time, definitive programming eliminates or relieves most of the limitations of spreadsheets. With respect to (L1), instead of a fixed size table of cells, definitive programming allows unlimited number of variables. With respect to (L2), our current definitive system can incorporate the definitive notations Scout and DoNaLD which can be used as tools for developing graphical user interfaces. With respect to (L3) and (L6), definitive programming is not confined to writing a set of

definitions; the ADM is an example of a definitive programming language which has higher-level control over sets of definitions. Not much improvement to (L4) and (L5) though except providing logging facility in our system.

To summarise, definitive programming retains most of the advantages and eliminates or relieves most of the limitations of spreadsheets. Therefore, according to Hewett's argument, definitive programming is a competitive tool for rapid prototyping.

3.1.2.3. Cognitive Dimensions

Green ([Green89]) generalises Streitz's observation of 'writing is rewriting' ([Streitz88]) to 'design is redesign' and 'programming is reprogramming'. Since a programming exercise involves frequent re-evaluation and modification, the amount of information that can be extracted from the notation with respect to re-evaluation and modification becomes important. Some cognitive dimensions for programming notation design are discussed in [Green89]:

1. *Hidden/Explicit dependencies* – how easy is it to cross-reference related information?

2. *Viscosity* – how easy is it to make localised changes? For example, how easy is it to insert a new formula into a spreadsheet cell?

3. *Premature Commitment* – how easy is it to develop a program in a mental generative order?

4. *Role-expressiveness* – how easy is it to infer the roles of different parts of a program from the program fragments themselves?

5. *Hard Mental Operations* – how easy is it to understand the individual programming constructs? For example, are there constructs such as the eval and quote functions in Lisp and pointers in C that are particularly hard to understand?

Green observes that a typical object-oriented programming system, Smalltalk-80, does not score very well under the testing of the above dimensions. It seems, on my evaluation, that definitive programming may get a higher score.

Hidden/Explicit Dependencies – In a definitive script, dependencies can be extracted from the definitions easily. Take EDEN as an example. Although the EDEN environment does not disclose long range dependency, the '?'-command (query command) does provide local information about both forward and inverse dependency.

Viscosity – Green observed that inserting a new formula into a spreadsheet cell is very simple but actions that entail rearranging the layout are quite another matter (for instance, introducing a new row may have the side-effect of corrupting the value rules associated with other rows). A definitive notation is similar to a spreadsheet in that assigning a new formula to a definitive variable is straightforward. Because definitive variables are not subject to the same geometric conventions and constraints as spreadsheet cells, new variables can be introduced very simply. At the same time, it should be noted that there is no operation on a definitive script that corresponds to introducing a row into a spreadsheet. Also, as explained in §3.1.1.2., the geometric conventions of a spreadsheet offer greater expressive power.

Premature Commitment – The order of definitions appearing in a definitive script is irrelevant, and so is the mental order of program design. Variables can be redefined at any stage, and in any order. In chapter 5, we show a top-down design strategy of a screen layout. On the other hand, we can, for example, incrementally add on new meters on the panel of the vehicle cruise control simulation in chapter 7 – this reflects a bottom-up approach of program design.

Role-expressiveness – In one respect, definitive notations show high role-expressiveness. Definitive notations have application-specific underlying algebras. The role of the definitions is implanted in the design of the underlying algebra. In another respect, definitive notations are low in role-expressiveness. Since the order of definitions is unimportant, a definitive script can be difficult to understand. For instance, the definitions for the lamp on a table can be far away from the definitions for the table itself. Automatic reordering of definitions can only help to a limited extent. Seemingly the most natural way of sorting the definitions is sorting by dependency. However, this may not be the best way depending on the occasion. In one context, one may like to group all the definitions concerning screen layout together but in another to group the definitions about the visualisation of a variable together no matter how many different definitive notations are involved. Nevertheless, if a definitive script is properly organised, because of the application-specific nature of definitions, definitive notations should attain a very high role-expressiveness.

Hard Mental Operations – Definitive programming, at its present stage, still shares the simplicity a spreadsheet enjoys. The particular examples of hard mental operations cited by Green, such as pointers and indirection in C, eval and quote functions in Lisp, are absent. Whether there are any hard mental operations requires further research by the cognitive psychologists.

In summary, the virtues of spreadsheets with respect to the cognitive dimensions suggested by Green are retained by definitive notations whilst some of the disadvantages of spreadsheets have been overcome.

3.2. Document Preparation Software

The most prominent use of dependency in a document preparation system is in the definition of styles. In a style-based document preparation system, style information

can be associated with individual characters, paragraphs, sections or the whole document [JB88]. Changes in the definitions of the styles will affect the presentation of the text; for a style-based WYSIWYG (What You See Is What You Get) editor, these changes will be reflected interactively on the screen. A typical style-based WYSIWYG editor is Microsoft Word. In Microsoft Word, a new style can be defined upon an existing style. Using this thesis as an example (as it is prepared using Microsoft Word), the first paragraphs following the headings are in Normal style; the subsequent paragraphs are in NL style. Normal and NL are defined as:

Normal :- Font: Times 12 Point, Justified, Line Spacing: 24pt, Space Before 12pt

NL :- Normal + Indent: First 0.5in

If the line spacing of the Normal style is redefined to 12pt (single line spacing, say for printing the first draft), the paragraphs with the NL style also become single line spaced.

Lilac [Brooks91] is another style-based editor. Lilac is both WYSIWYG and language-based. Because it is WYSIWYG, a change of style takes immediate effect on the screen, which is a close approximation to the printed output; because it is language-based, a user have more flexibility in defining styles. By providing similar programmability as in Troff and Tex [Knuth84], Lilac allows the user to define complicated styles such as might be used in a periodic table. Lilac has data types and operations specific to document preparation. Its data types are Box, Hglue, Vglue (horizontal and vertical glue between boxes), Hlist, Vlist (horizontal and vertical lists of objects), Num (number), Bool (boolean), Family (font family), Face (typeface) and Font. There are basic operations defined on these data types and user-definable operations can be defined on top of these basic operations. A document is generated by applying the operations (styles) to the text of the document.

It is clear from the form of Microsoft Word style definitions that style definitions can be regarded as definitions in the definitive programming sense. A redefinition of the base style has indivisible effects on all the styles directly or indirectly

defined upon it. Lilac goes further to describe an underlying algebra for styles. This further justifies the claim that many style-based document preparation systems are, theoretically speaking, definitive notations for document preparation.

3.3. Graphics Editors and User Interface Tools

Conventional graphics editors, such as MacDraw, do not make use of the dependency between objects. In graphics tools at the research level, the need for dependency is more commonly recognised. L.E.G.O. is a construction-based drawing language in which an object can be constructed with reference to other existing objects [FP89, FP88, FP86]. Since L.E.G.O. is an imperative language, the use of dependency information between objects has not fully exploited. There are however other graphics editors like Ded [Jeet87], GIPS [CFV88] and NoPumpG [Lewis87] which use dependency in a more declarative manner. The re-construction of an object will update the position or the shape of those objects dependent on it. Some graphical user interface (GUI) tools like ThingLab [BD86], Coral [SM88], RENDEZVOUS [Hill92] and Views [Pemberton92] also use constraints to establish links between graphical objects and between graphical objects and application-generated data.

The basic task of a graphics editor is to enable the user to manipulate and visualise the abstract model of a graphical image. [Beynon85], [Beynon88a] and [Beynon89] argue that *definition* is a suitable abstraction for the task. In fact, both Ded and NoPumpG use uni-directional relationships (*definitions* in our terms) for constructing the abstract model. The kinds of relationships within these graphics systems by-and-large concern the geometry of individual graphical objects. In addition to supporting geometrical relationships, NoPumpG incorporates a system clock that can be used to describe the relationship between graphical objects and time. For this reason, NoPumpG is better known as a tool for animation than as a graphics editor.

Graphical user interface tools resemble graphics editors in that they are both concerned with visualisation and manipulation of data. While the data involved in a

graphics editor is the abstract model of the graphical objects themselves, the data to be visualised and manipulated in a GUI comes from a separate application. Therefore, the relationships between visual object and the application data have also to be addressed in GUI. [BY90] clearly identifies that the visualisation process has characteristics similar to a mechanical linkage. In a mechanical linkage, a change in position in the input end immediately changes the position of the output end; the change of a view of an abstract model should always be synchronised with the change of the abstract model itself. Many GUI tools, such as those mentioned above, use constraints to link the abstract model and its views. The reason of using constraints is not only because there is a close relationship between abstract model and view but also because the interpretation of input is closely related to both abstract model and view. The multi-directional relationship described by a constraint enables a change of a view to effect a change in the abstract model. In our paradigm, we recognise the close relationship between model, view and control but reject models in which one can hurt other person by hitting his shadow. In our method, definition is the link between model and view. The uni-directional nature of definition perfectly describes the relationship between model and view. An input is interpreted in the context of the view but will directly affect the model. The view is updated indivisibly with the change of model. More detailed discussion on our way of handling input will appear in Chapter 7. In short, the GUI paradigm we are employing can be depicted as follow:

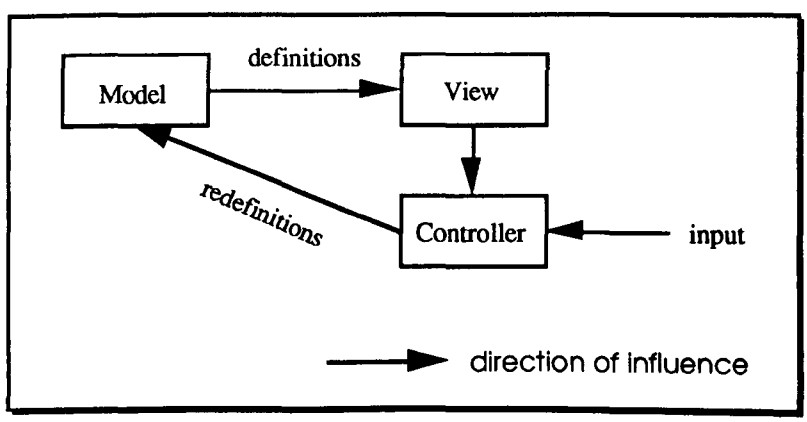


Figure 3.1: Graphical User Interface Mechanism in Our Systems

The view is linked to the model via a set of definitions. The controller interprets the input based on the information obtained from the view definitions and manipulates the model by redefinitions.

3.4. The Make Utility

The last kind of the software to be discussed which make uses of dependency is the make utility. Make is originally a standard UNIX utility for file management, and is now widely used in the PC community. In make, the dependency between files can be specified together with commands for updating the files. A typical use of make is for compilation of programs. The following is a simple example of a makefile:

```
1. calc: main.o function.o
2.         cc -o calc main.o function.o

3. main.o: main.c
4.         cc -c main.c

5. function.o: function.c function.h
6.         cc -c function.c
```

Listing 3.1: An Example of Makefile

This makefile is intended for the compilation of a calculator program. The program is written in two modules, main.c and function.c. Function.c includes a header file function.h. Since the C compiler supports separate compilation, two more files, main.o and function.o, also need to be kept up-to-date. While lines 1, 3 and 5 in the example specify the dependency between the files, lines 2, 4 and 6 specify the way in which the files are to be updated. In UNIX, there is a set of attributes associated with every file. These include the last access time and the last modification time. When make is invoked, it checks the status of the files on the dependency lists. If any file on the dependency list has its last access time equal to its last modification time (which means that that file has recently been updated), the target file has to be updated according to the specified rule. At the beginning of a make session, make generates a dependency tree

and checks the status of the files starting with the leaves of the tree. In this way, duplicate updating of the same file can be prevented. This mechanism is similar to the implementation of our system except that our system is continuously executable. What corresponds to the dependency tree in the case of definitive variables is constantly maintained rather than being regenerated in every evaluation.

Unlike definitions in which data dependency can be inferred from the definitions themselves, the file dependency has to be stated explicitly in what is called a makefile. This is because the commands for file maintenance do not generally disclose which are the source files and which are the target files. The command in line 6, for instance, gives no indication that function.h is one of the depending source and function.o is the target file. This dependency is implicit in the content of function.c and the conventions of the C compiler. However, different commands have different conventions. There is no general rule for inferring the file dependency from a command.

3.5. Theoretical Framework

We have examined some commonly used software tools in this chapter. In these tools, dependency plays a significant role. Perhaps it is their use of dependency that makes them so popular. It is however worth noting that, despite having different histories of origin, these tools express dependency in very similar ways. As we have mentioned in the discussion, the underlying principle behind such software is not dissimilar to that of definitive programming. It is our belief that dependency is not only beneficial in individual applications but is applicable to broader issues of software development. Our research in definitive programming is an effort to develop a theoretical framework of programming that is based on dependency.

4

The Scout Notation

The simplest way of applying the Definition-based State-Transition (DST) model is to develop definitive systems similar in kind to simple spreadsheet software. In such systems, the only source of transition of state is via direct redefinition by the user. Definitive systems differ in the data types and operators that they employ in their definitions. Definitions of specific data types and operators are particularly suitable for specifying states in different applications. Each such notation is called a *definitive notation.*

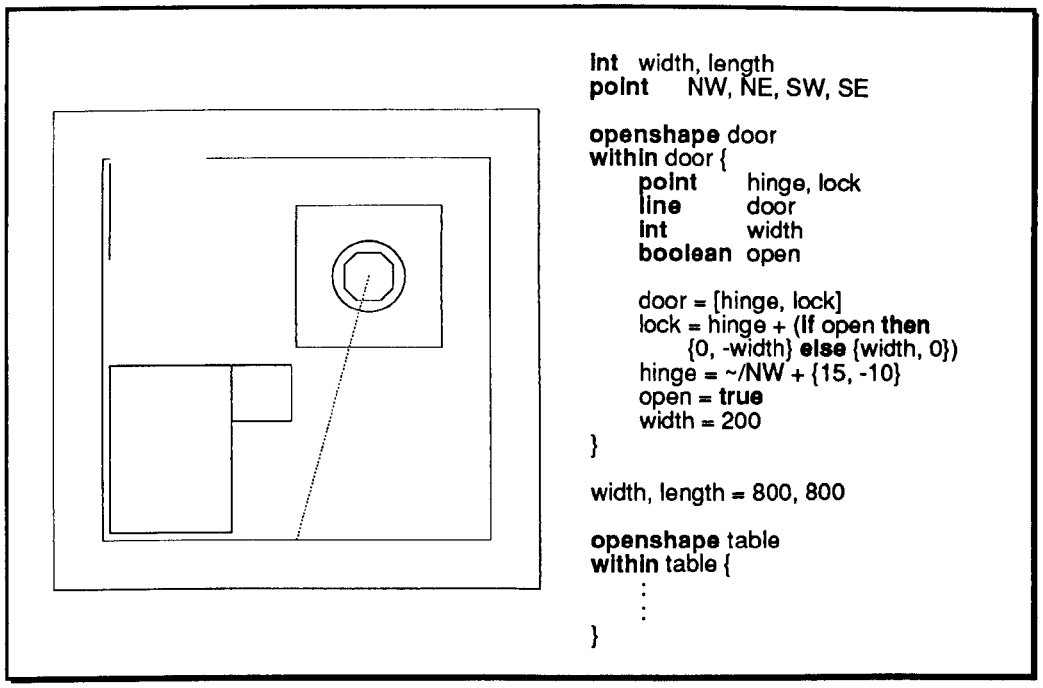
As explained in the last chapter, a typical definitive notation differs from an electronic spreadsheet in that variables in a definitive notation have their own names and are independent entities. Variable names in a spreadsheet on the other hand are signified by their geometric locations on a table of cells. As in a spreadsheet, a variable in a definitive notation can be defined explicitly by a value or implicitly by a formula in terms of other variables. A collection of declarations of variables and their definitions is called a *definitive script.*

Since every definitive notation has its own application domain, each definitive notation has its own right of existence. This chapter describes a definitive notation called *Scout* which I have designed and implemented. The development of this notation has made important contributions to our understanding of the modelling property of definitive notations and to the integration of definitive notations, but this chapter only describes Scout as a definitive notation for the fulfilment of its original function – describing the screen layout – and leaves the discussion of its other contributions to later chapters. The aim of this chapter is to introduce the Scout notation for further discussion later. Both the design and the implementation of the Scout notation will be described.

4.1. The DoNaLD Notation

The Scout notation is unique amongst the existing definitive notations in that it provides complementary display information for other definitive notations. To appreciate this role of the Scout notation fully, it is best to briefly introduce another definitive notation. The notation to be introduced is DoNaLD.

DoNaLD is a definitive notation for two dimensional line drawing. The notation was defined in 1986. The full specification of DoNaLD can be found in [BABH86]. The first prototype of DoNaLD was developed by Edward Yung in 1987, but not all features in [BABH86] were implemented. In subsequent enhancements ([Chan89, Parsons91]), some new data types and operators have been introduced, yet some features in the original specification remain unimplemented. However, the conceptual framework for definitive notations is sufficiently brought out by the current DoNaLD prototype. An illustrative example of DoNaLD, which is used many times in our publications, is a description of a room. Figure 4.1 shows a part of the room specification and the graphical visualisation of the entire room.



```
int  width, length
point    NW, NE, SW, SE

openshape door
within door {
     point      hinge, lock
     line       door
     int        width
     boolean    open

     door = [hinge, lock]
     lock = hinge + (if open then
          {0, -width} else {width, 0})
     hinge = ~/NW + {15, -10}
     open = true
     width = 200
}

width, length = 800, 800

openshape table
within table {
     :
     :
}
```

Figure 4.1: DoNaLD Script of a Room

In the current DoNaLD implementation, there are eight data types: *integer*, *real*, *boolean*, *point*, *line*, *circle*, *ellipse* and *shape*. DoNaLD variables are typed variables. A *point* variable corresponds to a point on the screen; a *line* variable corresponds to a line. The same is true for *circles* and *ellipses*. A *shape* variable corresponds to a group of elementary graphical elements on the screen. Since *integer*, *real* and *boolean* are not graphical items, variables of these kinds do not have any visible form on the display.

The geometric aspects of the graphical items are determined by the values of the variables. The value of a *line* variable, for instance, prescribes the location of the two end-points on the screen. The DoNaLD system employs an orthogonal Cartesian coordinate system: the lower bottom corner of the screen is the origin, the x-axis goes from left to right and the y-axis goes upwards in the scale of one unit per pixel.

No part of the DoNaLD notation is dedicated to the presentation of the graphical items. That is to say, there is no formalised way of controlling, for example, the colour of a line in DoNaLD. Our convention is that a line in DoNaLD will appear to be a solid black line with unit thickness. In our current prototype, however, attributes can be

associated with a variable to control the line style (if it is a line) and colour. These attributes are also defined in the definitive style. Redefining the attribute of a variable will automatically take effect on the screen.

4.2. Motivating the Design of Scout

The name Scout is an abbreviation of 'SCreen layOUT'; the Scout notation is a definitive notation for describing screen layout. It is a notation concerning how to display information, i.e. a definitive state, on the screen. There are two main motivations for designing the Scout notation: to present the definitive state in a user-specified manner and to supplement the display information for other definitive notations.

4.2.1. Visualisation of State

The value of a variable is typically represented in a computer by a sequence of 0's and 1's. This binary representation is not particularly useful in high-level programming. Values need to take another form of representation in order for the user to understand. The value of a variable storing the room temperature has one thousand and one presentations: angular displacement of a meter, length of a bar, seven-segment digital readings, colour code, and so on. The choice of presentation method is up to the designer and sometimes can be selected by the users.

The role of definitive notations is to describe state. The external presentation of the internal state is determined by the implementation of the definitive systems. Current systems exploiting dependency typically use different ways of presentation for different types of variables. For example, in a spreadsheet, there may be cells (variables) that display textual strings and other cells that display numerical values; their values will be represented on the display by strings of characters and strings of digits respectively. DoNaLD provides other examples: a *point* variable with value {x, y} will appear as a dot on position x steps right and y steps up from the origin; a *line* variable with value

[{x1, y1}, {x2, y2}] will be represented by (what appears to be) a collinear set of dots on the screen. Of course, the string "{10, 20}" or *a dot on screen* are alternative presentations of the same value, but obviously, displaying a dot in a particular location may be much more appropriate than displaying a string anywhere on the screen. However, this is not enough to justify the way current definitive systems operate: fixing the presentation formats of the variables during the design and implementation of the system.

We do not always want values of the same type to have the same presentation. This becomes obvious when we compare the representations of the same data types in two different notations. Take the *integer* type as an example: an integer value is not displayed on the screen in DoNaLD, but in another context, such as a spreadsheet, a string of digits will be displayed in a cell that records an integer value. Even within the same definitive notation, there are cases when we would like to see different (variants of the same) representations for the same type of values. Lots of examples can be found from the DoNaLD notation: in a floor plan, we may like the line denoting a wall to be thicker than the line denoting a door; we may like to use dotted line to represent some flexible linkage, say an electric cord; or we may like to define a box whose corners are specified with reference to the centre of the box but we do not want to see a dot in the middle of the box (i.e. the visualisation of the variable of the central point).

The Scout notation addresses the problem of presentation of data by using definitions to describe the output formats of a variable. With definitions, a persistent link between the internal model and its external representation is achieved. The observed changes of variables can be synchronized with internal state changes. Scout definitions are therefore performing a function analogous to the format rules in the spreadsheets. In fact, Scout allows more flexible control over the output format.

4.2.2. Assumptions of Definitive Notations

Most hardware primitives cannot be altered by software. For example, the location of every pixel on the physical display is determined by the manufacturer. The correct interpretation of a definitive notation has to take into account the particular hardware characteristics. The user of a definitive notation cannot interpret a definitive script precisely unless he knows about the assumptions made by the notation designer. There has to be in effect a "contract" between the notation designer and the programmers – perhaps specified external to the system through a manual.

The screen described in the Scout notation is actually not the physical screen. It is, in principle, describing an imaginary screen. As for other definitive notations, there is a mapping from the imaginary screen to the physical screen. In the design of Scout, the intention is to create a close correspondence between the two screens – for instance, a point in the imaginary screen should map to a point on the physical screen so that Scout can use the maximum resolution of the display unit. However close the correspondence is, clarification beyond the scope of the notation is still required. For instance, a point in a TTY screen has a slightly different interpretation from a point in a workstation – a point in a TTY screen is large enough to display a character but a point in a workstation cannot.

When writing programs, the programmer may prefer having computer hardware with particular physical characteristics. It is our hope that we can, by means of the definitive notation Scout, create an interface between the programmer's preferred environment and the environment provided by the actual machine. This can be illustrated with reference to DoNaLD.

Writing a DoNaLD specification has to take into account assumptions about the physical output device in use. Such assumptions include the size and the resolution of the screen and the location of the origin. Consider, for instance, the task of specifying

a square in the middle of the screen. Listing 4.1 is a plausible specification if the origin is located at the centre of the screen.

```
openshape square
within square {
      line   N, E, S, W
      integer size
      size = 100
      N = [{size,size}, {-size,size}]
      E = [{size,size}, {size,-size}]
      S = [{-size,-size}, {size,-size}]
      W = [{-size,-size}, {-size,size}]
}
```

Listing 4.1: A DoNaLD Description of a Square

If the origin is located at the bottom-left corner, shifting has to be done in order that we can see the whole square. A variable centre may be added to Listing 4.1 to do the shifting (see Listing 4.2). However, the value of centre (say {300, 200}) cannot be written down without prior knowledge of the size of the display (say 600×400). Moreover, the output on the display will genuinely be a square only if the vertical and the horizontal axes carry the same resolution.

```
openshape square
within square {
      line N, E, S, W
      integer size
      point centre
      size = 100
      centre = {300, 200}
      N = [{size,size}+centre, {-size,size}+centre]
      E = [{size,size}+centre, {size,-size}+centre]
      S = [{-size,-size}+centre, {size,-size}+centre]
      W = [{-size,-size}+centre, {-size,size}+centre]
}
```

Listing 4.2: Another DoNaLD Description of a Square

Scout is a definitive notation for describing screen layout. It enables us to put down the assumptions about the required display so that the programmer can work on an imaginary screen that suits his purpose. Listing 4.3 illustrates how the Scout notation can be used to control the realisation of an image onto the physical screen. The top half of Listing 4.3 is the DoNaLD description of a square defined in a preferred

environment (i.e. Listing 4.1); the bottom half shows a Scout window in which the mapping between the preferred environment and the actual display is defined. The first three attributes of the specification of the sqr window – *type*, *box* and *pict* – specify that the DoNaLD picture "don" is displayed in the region prescribed in the box with the two opposite corners {0, 0} and {600, 400}. The attributes *xmin*, *ymin*, *xmax* and *ymax* describe the coordinate system within this window. The window sqr can display the region bounded by the lines x = xmin, y = ymin, x = xmax and y = ymax. In Listing 4.3, *xmin*, *ymin*, *xmax* and *ymax* are defined in such a way that the origin of the DoNaLD picture corresponds to the centre of the window and one unit in the DoNaLD picture corresponds to one pixel in the actual screen display.

```
%donald                    // beginning of DoNaLD script
viewport don               // name of the DoNaLD drawing
openshape square
within square {
      line N, E, S, W
                integer size
                size = 100
                N = [{size,size}, {-size,size}]
                E = [{size,size}, {size,-size}]
                S = [{-size,-size}, {size,-size}]
                W = [{-size,-size}, {-size,size}]
}
%scout                     // beginning of Scout script
window sqr = {
      type: DONALD,
      box:  [{0,0}, {600,400}],  //geometry of the window
      pict: "don",         // name of the DoNaLD picture
      xmin: -300,          //
      ymin: -200,          // geometry of the imaginary
      xmax: 300,           // DoNaLD display
      ymax: 200            //
}
```

Listing 4.3: A Sample Scout Fragment

With Scout, other definitive notations, such as DoNaLD, can be thought of as working with an imaginary screen that has idealised properties such as unbounded size, unlimitedly fine resolution and infinite number of colours. This imaginary screen concept is very important in ensuring that models can be built with minimal hindrance.

The output of the definitive script is obtained through a mapping from the imaginary screen to a physical screen. The actual output is different from the imaginary one both because there are constraints in the physical screen and because we may like to set some limits on the actual output (for example, to show the region bounded by the lines x = 0, x = 1000, y = 0 and y = 1000 of a DoNaLD picture in a particular region of the physical screen with resolution of 4 square units per pixel). Definitive notations like DoNaLD and ARCA do not give the user control over how the imaginary screen maps to the physical screen. For this reason, assumptions about the mapping have to be fixed at the implementation stage when these notations are to be used on their own. The Scout notation, on the other hand, presents supplementary information about the mapping for other definitive notations so that more control over the output is possible.

4.3. The Design of Scout

A preliminary design for the Scout notation was described in my final year undergraduate project report [Yung88]. In its original form, Scout was designed for laying out text; it was subsequently enhanced to interface with DoNaLD and ARCA.

The choice of the Scout notation primitives is based on the assumed nature of the screen and the manner in which it will be used in particular applications. For instance, a newspaper layout may require multi-column format, whilst rectangular boxes suffice for a typical window-based application. For these tasks, high resolution graphical display is a reasonable assumption. Provision for colour display is also preferred.

Complicated drawings are assumed to be handled by other definitive notations. In fact, the purpose of designing different definitive notations is to ensure that different kinds of application can be addressed in appropriate notations. DoNaLD, CADNO [Stidwill89] and ARCA are some definitive notations designed for describing different kinds of graphics. Other definitive notations for describing graphics are also

conceivable. It is not our intention and it is not appropriate to use Scout to describe every single detail of what the screen will display. Scout, therefore, does not provide a set of drawing primitives but is intended to deal with simpler tasks such as screen layout, scaling and other operations at the pixel level. Scout's special role is to specify where and how these images described by other definitive notations are displayed. In principle, the role of Scout should be confined to coordinating the use of different definitive notations to form a display, but because there is no definitive notation for displaying text so far and text is almost indispensable in any serious application, a part of the Scout notation is concerned with the display of text strings. However, it is intended to develop other definitive notations to deal with more complicated text displaying tasks such as those encountered in desktop publishing or word processing applications.

4.3.1. The Window Data Type

Layout design in Scout is based on the answers to the following three questions: What are the things to be displayed? Where are they to be displayed? And how are they to be displayed? This naturally leads to the concept of the Scout *window* data type. The type *window* is a union of subtypes: one subtype is designed for each definitive notation. Each subtype has a number of fields. The number of fields and the types of the fields may vary depending on which notation this subtype refers to. Generally speaking, a *window* should have fields that define

1) which definitive notation is concerned;
2) what information (e.g. which DoNaLD picture or which text string) is to be displayed;
3) the region of the screen onto which the required information is mapped;
4) the supplementary information that that definitive notation needs.

In the current design and implementation, there are three types of windows – the text window, the DoNaLD window and the ARCA window.

Text Window		
field name	***type***	***description***
type	content[1]	Must be the value `TEXT`
string	string	The string to be displayed
frame	frame	The region in which the string is shown
border	integer	Width of the border of the boxes of the frame
alignment	just[2]	`NOADJ`, `LEFT`, `RIGHT`, `EXPAND` and `CENTRE` are the possible values to denote no alignment, left justification, right justification, left and right justification and centre of the text inside each box in the frame
bgcolour	string	Colour name for the background colour of the text
fgcolour	string	Colour name for the (foreground) colour of the text

where point = integer × integer

box = point × point

frame = *list of* box

DoNaLD Window		
field name	***type***	***description***
type	content	Must be the value `DONALD`
box	box	The region in which the DoNaLD picture is shown
border	integer	Set the border width of the bounding box
pict	string	The name[3] of the DoNaLD picture
xmin	point	Show the portion of the DoNaLD picture bounded by the points (xmin, ymin) and (xmax, ymax)
ymin	point	
xmax	point	
ymax	point	

---

[1] The type *content* is currently a set { `TEXT`, `DONALD`, `ARCA` }.

[2] The type *just* is { `NOADJ`, `LEFT`, `RIGHT`, `EXPAND`, `CENTRE` }.

[3] There is no picture name specified in the original DoNaLD notation, but there is now a statement

`viewport` *name*

required before the DoNaLD definitions to identify which picture these definitions are defining.

• The ARCA window is exactly the same as the DoNaLD window except that the type of window should be declared to be `ARCA`.

Comments on the window data types:

1) The definitions of the window subtypes above are very simple. Many more attributes, such as background pixmap, font of string and so on might be used to control the appearance of the windows. Actually, there is no real reason why those attributes cannot be included into the Scout windows. The attributes listed above are chosen simply because they are the most commonly used ones. Introducing more attributes reduces the number of defaults that are built into the interpreter, giving the user a higher degree of control over window specification.

2) There is no formal restriction on how to define a region. Nevertheless, the choice of method should be governed by the nature of application. The three available subtypes have already employed two ways of defining regions. In a DoNaLD or ARCA window, a region is defined by a box, whereas in a text window, a region is defined by a list of boxes. A single box is good enough to frame one picture but a list of boxes is required if a long passage of text is to be displayed in multiple columns. Other methods of defining regions might also be employed. For reference, the X Toolkit [MAS88] is primarily designed for rectangular windows; PostScript [Adobe85] defines an arbitrary shaped region by a closed path; a bitmap is commonly used to define a small region such as the shape of an icon.

3) Notice that there are almost no fields in common between the graphics window subtypes and the text window subtype, so the type *window* may be better understood by the abstract formula

$$\text{window} = \text{region} \times \text{content} \times \text{attributes}$$

rather than by a concrete set of fields.

4.3.2. The Display Data Type

A *display* is a collection of windows. Because the windows may overlap, there is a partial ordering among the windows. For simplicity, a *display* is defined to be a list (total ordering) of *windows*.

In general, a *display* variable represents a conceptual screen; only the distinguished *display* variable *screen* denotes the physical screen. There are simple rules for mapping the variable *screen* onto the physical screen:

1) the origin is defined at the top-left corner of the physical screen;

2) the x-coordinate counts from the origin to the right, one unit per pixel;

3) the y-coordinate counts from the origin to the bottom, one unit per pixel.

It is obvious from the mapping rules that the interpretation of the Scout notation, unlike other definitive notations, is hardware dependent. The same script of Scout definitions may have a slightly different look on a monitor with different resolution and aspect ratio.

4.3.3. Other Data Types and Operators

Because there is a great flexibility in the design of the window data type, the set of data types and operators in Scout may be extended in the future. There are, however, some essential data types in Scout: *integer*, *point*, *window* and *display*. Associated with them are basic operators for integer arithmetic, vector manipulation, list manipulations, construction and selection. The following table shows the basic Scout operators and functions for the four essential data types. All the operators of the Scout notation can be found in Appendix A.

Operators:	+, –, *, /, % (remainder), – (unary minus)
Meaning:	Normal integer arithmetics
Example:	10 % 3 gives 1
Constructor:	{*x*, *y*}
Meaning:	Construct a point
Example:	{10, 20} is a point with x-coordinate 10 and y-coordinate 20
Operators:	+, –
Meaning:	Vector sum and vector subtraction
Example:	{10, 20} – {20, 5} gives {–10, 15}
Selector:	.1, .2
Meaning:	Return the 1st (x-) coordinate and the 2nd (y-) coordinate resp.
Example:	{10, 20}.1 gives 10
Constructor:	{*field-name*: *formula*, *field-name*: *formula*, ..., *field-name*: *formula*}
Meaning:	Constructing a window
Example:	{ type: DONALD, box: b, pict: "figure1" }
Selector:	*.field-name*
Meaning:	Return the value of the field
Example:	{ type: DONALD, box: b, pict: "figure1" }.box gives b
Constructor:	< *W1* / *W2* / ... / >
Meaning:	Constructing a display, if *W1* and *W2* overlap, *W1* overlays *W2*
Example:	< don1 / don2 >
List function:	insert(*L*, *pos*, *exp*)
Meaning:	Insert the expression *exp* in the position *pos* of list *L*
Example:	insert(<w1, w2, w3>, 2, new) gives <w1, new, w2, w3>
List function:	delete(*L*, *pos*)
Meaning:	Delete the *pos*th element of list *L*
Example:	delete(<w1, w2, w3>, 2) gives <w1, w3>
Operator:	if *cond* then *exp1* else *exp2* endif
Meaning:	if *cond* gives non-zero value (true) then returns *exp1* else returns *exp2*, in this context,*exp1* and *exp2* must have the same type.
Example:	if 1 then "Open" else "Close" endif gives "Open"

As mentioned, text layout should ideally be described by another definitive notation. Since that notation does not exist, part of the Scout notation is designed for simple text layout. To this end, Scout incorporates a *text window* subtype. This text window subtype differs from other window subtypes in that the content of the text window subtype is a string defined within Scout rather than a virtual screen prescribed outside Scout by another definitive notation. As a result, *string* becomes one of the Scout data types.

Associated with the *string* data type is a set of operators useful for displaying a text string. String concatenation (//), string length function (strlen), sub-string function (substr) and integer-to-string conversion (itos) are the basic Scout string manipulation functions. There are two postfix operators – .r and .c – which are specially designed. Since the basic geometric unit in Scout is the pixel but the size of a block of text is more conveniently specified as "number of rows by number of columns", it is convenient to introduce functions returning the row height and the column width in pixels. .r is the function meaning "multiply by the row height" and .c is the function meaning "multiply by the column width". These functions are appropriately represented by postfix operators because they work very much like units. For example, {10.c, 3.r} refers to a point 3 rows down and 10 columns right to the origin. A similar consideration influences the design of a *box*, a data type for defining regions. The region associated with a box is sufficiently defined by its top-left corner and its bottom-right corner, and this is a convenient method of definition in the case of graphics. For a block of text, however, the bounding box is more conveniently defined by specifying the top-left corner and the dimensions of the box in terms of number of rows and columns. For instance, [{0, 0}, 3, 10] refers to a box with the origin as its top-left corner which is suitable for displaying three rows by ten columns of text. More examples of this kind can be found in the Jugs example in the next chapter.

Because displaying a string is different from displaying an image, the way of specifying a region for displaying text is different from that for displaying an image. As attested by the fact that the earlier releases of X Window system version 11 had only primitives for creating rectangular windows, a simple box is adequate for display purposes in most applications. But, when considering the possible application of text display to desktop publishing, we know that a piece of text may be displayed across several regions; defining a region by a box is simply not sufficient.

One proposal is to define a region by a closed path (a wire-frame). This can create an arbitrary shape but this will also cause problems filling in the string. Consider the following *frame*:

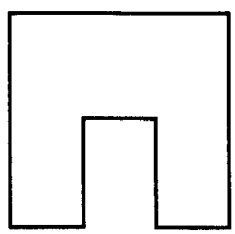


Figure 4.2: A Non-rectangular Region

There are two possible and equally natural ways of filling in a string as illustrated in Figure 4.3a and 4.3b.

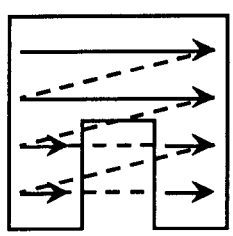


Figure 4.3a

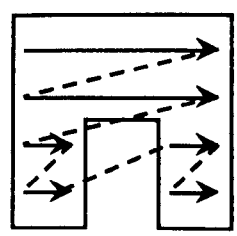


Figure 4.3b

Therefore, this way of defining region leads to ambiguity. Another ambiguity arises when a frame comprises two or more discrete closed paths. It is then unclear which closed path the string should fill first. Our solution is to divide an arbitrary shape into subregions, each of which is a box. The definition of a region will then be a ordered

list of boxes. For example, the region depicted in Figure 4.4 is interpreted in such a way that a string should be filled in the first box first, then the second, then the third.

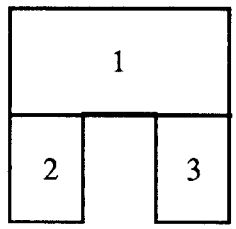


Figure 4.4: A Way of Partitioning a Non-rectangular Region

This "frame = list of boxes" definition of region is not perfect. For instance, if two boxes overlap (which may depict the overlapping of two sheets of the same document), which box should be put on top is still ambiguous. However, except for serious desktop publishing, this definition of region should be adequate for most applications.

4.4. The Implementation of Scout

Scout is implemented using a previously developed definitive language, EDEN [Yung89]. EDEN – an abbreviation of "Engine for DEfinitive Notations" – is intended to assist the implementation of definitive notations, though it is also a general-purpose programming language in its own right.

A unique combination of language features makes EDEN the most suitable tool for implementing other definitive notations. EDEN has:

- C-like syntax and operators. That is, EDEN has a rich set of efficient programming structures and operators.
- list structure. Complex data types can be simulated using lists.
- user-defined functions and procedures. These are essential for simulating operators in the definitive notations.
- definitions. EDEN provides automatic maintenance of definitions. The values of the EDEN variables will always be kept up-to-date using the dependency

information implicit in the formulae of the variables. Writing a translator for translating definitions in a definitive notation to EDEN definitions effectively creates an interpreter for that definitive notation. But writing a translator is much simpler than writing a definition maintainer.

- actions. EDEN actions are procedures whose execution is triggered by the changes to specified variables rather than being invoked explicitly by another procedure or the user. In implementing Scout, the EDEN actions are used to update the external presentation of the changed values of variables. The function of the EDEN actions can be depicted in the following figure.

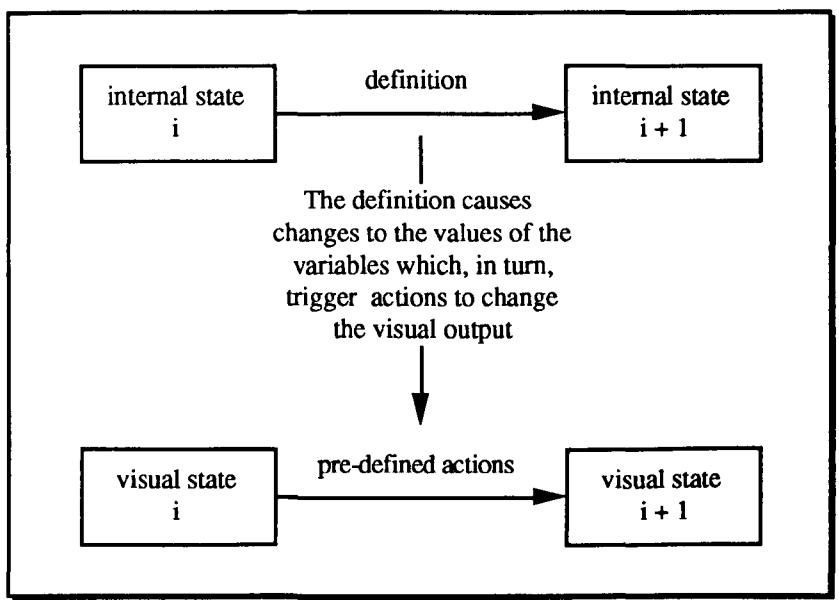


Figure 4.5: The Role of EDEN Actions in the Implementation of a Definitive Notation

As a simple illustration,

```
proc update_A : A
{
        writeln("A is ", A);
}
```

defines an action update_A. It will be invoked whenever A changes value. This action will print out a line expressing the new value of A.

The implementation task for Scout is to create a translator from Scout to EDEN. The translation is carried out according to the scheme to be discussed in Chapter 6,

where issues concerning the integration of definitive notations are considered. The implementation involves writing an EDEN library to simulate the Scout data types and operators and writing a program for translating Scout definitions to EDEN definitions.

The integer data type in Scout and the arithmetic operators have direct equivalents in EDEN. The point data type is an ordered pair (a list of two integers). The box is fundamentally a pair of points, there is however a need to write an EDEN function for the point-row-column form of defining boxes. This function will take in a point and two integers as its arguments and return a box. Both the frame data type and the display data type are lists. The window data type is simulated by a list of the union of all the attributes of all the window sub-types. These attributes are enumerated data types, which can be represented by integers, strings, a primitive data type in EDEN, or one of the data types already mentioned. In this way, all the data types are directly or indirectly supported by the EDEN primitives. Hence the translation of Scout definitions into EDEN definitions is fairly straightforward.

The last part of the implementation involves the writing of the actions for maintaining the visual display. There is only one variable in Scout – screen – that has a visual representation. This is because only the display variable screen represents the physical display screen. For this reason, only one EDEN action is needed in the implementation and it is triggered by the variable screen.

There have been two versions of the Scout implementation. The first version was developed on the SunView window system and second on the X Window system. The SunView version uses the WW library from Rutherford Appleton Laboratory [Martin87] so as to simplify the implementation. The WW library is chosen because it has a function to format a string within a box. The reason for redeveloping Scout for the X Window system will be discussed in detail in Chapter 6. Because the X Window system does not provide that string displaying function, a similar function has to be written. In fact, we have built a separate program to interface between EDEN and the

X Window system. In this interface program, commands like creation of a box, displaying a string in a box and other line drawing commands are available. This interface program will be discussed in more detail in Chapter 6.

The SunView version is an early version; it does not have provision for combining the use of several definitive notations; the only window subtype is the text window. In the X Window version, graphics described by other definitive notations can be mixed in the same screen display. As mentioned, the Scout interpreter is not intended to draw the details of the pictures of other definitive notations. Rather, the Scout interpreter generates an X-window for each Scout graphics window. Scout controls the size and the scaling factors of these windows and the DoNaLD and ARCA interpreter draw the details in them. In other words, DoNaLD and ARCA are drawing on a transformed space whose transformation is determined by Scout – this corresponds to what conceptually Scout should do.

5

Definition-Based State-Transition Models in Application

We have abstractly discussed the Definitive State-Transition (DST) model and its advantages in connection with exploratory software development in Chapter 2; we have also described the design of a definitive notation, Scout, based on the DST model in Chapter 4. In this chapter, we will look at how we can apply the DST model to develop a screen layout for an educational game called *jugs* using the Scout notation. Then we will discuss the advantages of using definitive notations in the context of software development.

We choose to study the Scout notation because it describes the screen layout directly. Of course, in some sense, other definitive notations do describe the screen display or a portion of it. For example, the ARCA notation is designed as a software

tool not only to manipulate a class of abstract combinatorial diagrams but also to *visualise* them. But if we were going to consider other definitive notations instead of Scout, extra effort would be required to explain the relationships between the object being modelled, the internal model (i.e. the definitive script), and the actual output. For example in DoNaLD, the object might be a room, the internal model would describe the relative positions of the furniture in the room, but where this floor plan is displayed on a screen, the scale of the floor plan, and so on are still to be decided. On the other hand, the output of Scout is exactly the thing to be described by the script. In this case, only the relationship between the screen output and the script needs our attention.

5.1. The Jugs Problem

Jugs is a simple simulation program originally developed by Townsend[1], that was first considered from a DST prospective in [BNRSYY89]. There are two jugs, A and B, with different capacities, `capA` and `capB` respectively. `capA` and `capB` should be relatively prime. One can choose an operation from a set of permissible menus at a time. The whole range of operations is:

1) fill Jug A,

2) fill Jug B,

3) empty Jug A,

4) empty Jug B, and

5) pour as much water from Jug A to Jug B or from Jug B to Jug A as the destination jug can hold.

The target of the game is to leave a specific amount of water in either of the jugs.

---

[1] The original version is written by Ruth Townsend for the BBC computers. It is distributed by the Chiltern Advisory Unit.

The programming principles necessary to implement the selection and activation of menu options using a definitive approach are beyond the scope of this chapter. They will be discussed in Chapter 7 and 8 respectively. The role of the Scout definitions is to present the values of the variables of interest to the users in a comprehensible way.

5.2. Modelling a Screen Layout Using Scout

When the term 'modelling' is used, we mean that we have already at least a mental picture, if not anything more concrete, of what the target looks like. There is a distinction between modelling activity and exploratory design. For example, in the case of screen layout, exploratory design is necessary when the final screen layout is not known. Bits and pieces may be added, deleted or modified from the intermediate implementations until the designer is satisfied. For the Jugs problem, the emphasis is on modelling rather than exploratory design since the screen layout is prescribed rather than designed from scratch. We are basically following the layout of the output from the original Jugs program by Townsend. Therefore, before we do any exploration on the screen layout design, we begin by modelling the original Jugs output using Scout.

In the following sub-sections we will first discuss the process of modelling a screen layout using Scout, then consider some advantages of definitive notation in the light of the modelling technique demonstrated by Scout.

After the screen layout is modelled in Scout, the designer may go on exploring the design. The advantages of Scout, and in general definitive notation, towards exploratory development of software are going to be discussed in section 5.3.

5.2.1. Screen Layout Modelling Process

There are three informal stages for developing a Scout description of a screen layout:

1. Develop an idea of what the screen display should look like. For example, Figure 5.1 is what the screen should display when the Jugs program is first

started. The colour of the menus represents their availability – black on white indicates a valid option.

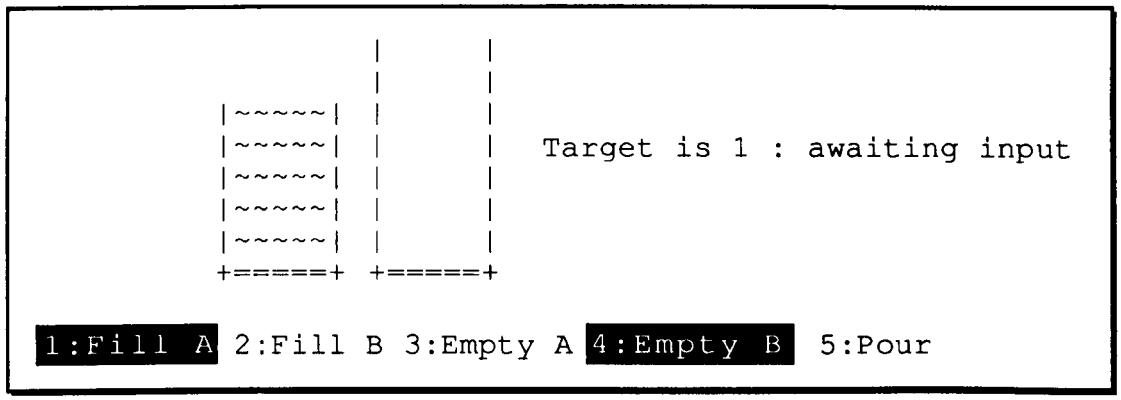


Figure 5.1: A Sample Jugs Output

2. Characterise the screen layout by identifying the common relationships in the screen layout. Figure 5.2 shows the design for the geometrical information of the Jugs output. Other characteristics such as the number of tildes required to fill up to the level contA (which is widthA × contA) can be identified as well.

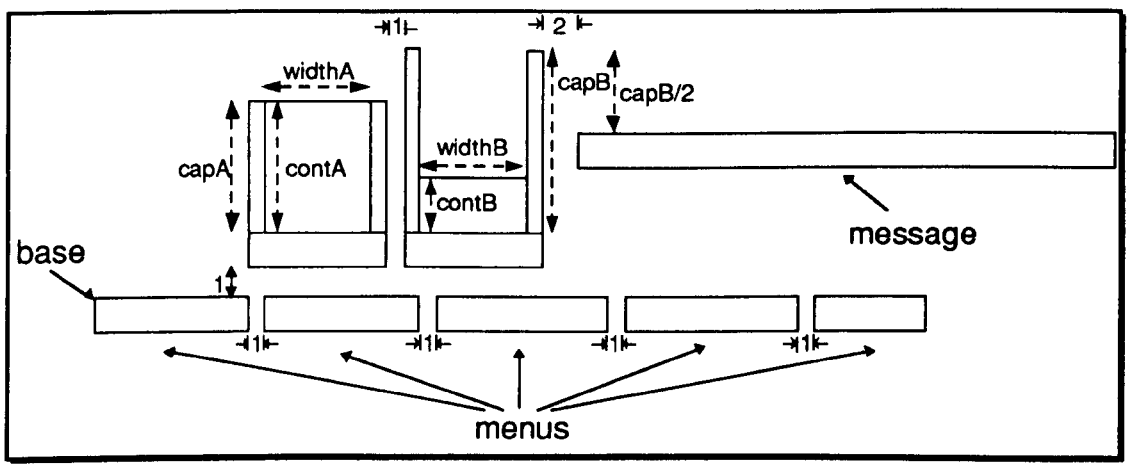


Figure 5.2: Screen Layout Design

3. The programming task is almost finished although we have not actually written down anything in the Scout notation! To finish off the work, this final step transforms the information obtained from the first two steps into the Scout notation. Listing 5.1 and Listing 5.2 show parts of the Jugs game screen layout in the Scout

notation. The complete Jugs example (Scout definitions for the screen layout and EDEN definitions for other part of the program) can be found in Appendix D.

```
point   base = {1.c, n.r} # 1 char-width(.c) right and n char-height(.r) down from origin
box     menu1box = [base, 1, strlen(pupilmenu1)]
# a box whose NE corner is at base, 1 char-height and strlen(pupilmenu1) char-width
box     menu2box = [menu1Box.ne + {1.c, 0}, 1, strlen(pupilmenu2)]
frame   jugAboxes = ([menu1box.ne+(0, –(2+capA).r}, capA, 1],
        [menu1box.ne+{(widthA+1).c, -(2+capA).r}, capA, 1],
        [menu1box.ne+{0, -2.r}, 1, widthA+2])
frame   jugBboxes = ([jugAboxes.2.sw + {2.c, -capB.r-1}, capB, 1], ...
box     contAbox = [jugAboxes.1.sw+{1.c, -contA.r}, contA, widthA]
box     messagebox = [jugBboxes.2.ne +{2.c, (capB/2).r}, 1, strlen(status)]
...
```

Listing 5.1: Definitions for Locations

```
string    backgroundi = validi ? "black" : "white"
          #reverse background  if option invalid
string    cA = repeatChar('~', widthA*contA)      #use '~'s to represent water level
string    jugA = repeatChar('|', 2*capA)//"+"//repeatChar('=', widthA)//"+"
...
window    menu1window = {
              frame: (menu1box);          string: pupilmenu1;
              bgcolour: background1;      fgcolour: foreground1
          }
# form a window by putting string pupilmenu1 (what to display) into a frame formed
# by a single box menu1box (where to display) displaying black on white or
# white on black depending the availability of the menu option (how to display)
window    capAwindow = { frame: jugAboxes; string: jugA }
window    contAwindow = { frame: (contAbox); string: cA }
...
display   screen = ( menu1window / menu2window / ...
                                / contAwindow / capAwindow / ... )
# screen represents  the physical screen; it displays the windows listed.
# If windows overlap, menu1window overlays menu2window etc.
```

Listing 5.2: Other Scout Definitions

This method of developing a screen layout is similar to writing a program in a traditional software development process; the first two steps are analogous to obtaining an (informal) specification whereas the last step is analogous to implementing the specification. Although the theme of this thesis is on exploratory software

development, the discussion in this section is not unrelated. The simplicity of the modelling method indicates how easily we can relate a definitive script to reality. This certainly helps the exploratory software designer to understand and make changes to the current design.

5.2.2. Special-Purpose Notation for Specific Task

The job of screen layout design is to decide where information should be placed and how it should be presented. The Scout notation restricts the areas allocated for displaying information to be rectangular or a group of rectangles. For this reason, the Scout notation permits only simple layout design. However, the design of the notation has already taken into account some assumptions of the characteristics of the display unit and the usual layout designs. For example:

i) *The Coordinate System*

The addressable points on a display unit normally form a grid. Moreover, Scout is only a notation for describing screen layout and is not a general graphics display notation. Therefore, the obvious choice of the Scout coordinate system is the Cartesian Coordinate System.

ii) *Area Allocation*

A window in Scout means a fixed region in which a piece of information is displayed. The region that can be allocated depends on the type of information to be displayed. Although no 2-D line drawing window appears in the Jugs example, Scout, at its present stage of development, can incorporate DoNaLD graphics, ARCA diagrams and text. If graphics is going to be displayed, the region must be a box. The following fields are significant in the definition of the window:

```
type: DONALD (or ARCA)
box:  b
pict: picture-name
```

where b is a box defining where the graphics should be displayed, and picture-name is the name of the DoNaLD or ARCA picture. If text is going to be displayed, the region is a frame rather than a box. A frame is used because it allows for more general display formats such as multi-column display and other irregular shaped regions. A text window should have the following fields defined:

```
type:    TEXT
frame:   f
string:  s
```

The declaration of text type is often omitted in a Scout program (for instance in the Jugs example) because windows are text windows by default. Note that the boxes of f are most conveniently defined by their top-left corners and by their dimensions (dimensions are expressed in terms of the number of characters in a row and a column).

iii) *Presentation of Information*

Again, what can be controlled depends on the type of information being presented. We can, for examples, shift and scale the image of the DoNaLD pictures and change the background colour of the window and the colour of the lines. For text, we can change its alignment, foreground and background colour.

iv) *Combining Windows*

In some cases, say a windowing system, windows may overlap. The Scout notation defines a display to be an ordered list of windows such that if there is overlapping, one window overlays another if it precedes the other in the list (cf. Listing 5.2). This presumes that it is never necessary to represent a situation such as Figure 5.3 where windows overlay cyclically.

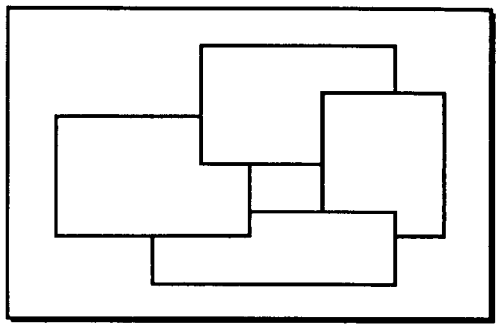


Figure 5.3: Cyclic Overlapping Windows

Although the Scout notation looks simple, its design has already involved a lot of assumptions about the nature of the physical displays, the types of application and the ways of denoting and manipulating information. For this reason, expressing the screen layout in Scout (step 3) is straightforward. Other definitive notations are also special-purpose notations. This means that the notations, including the data types and operators, are designed for particular application domains. This helps to give definitive notations high expressive power.

Moreover, using special-purpose notations reduces the learning time and the programming time of the programmer, increases the understandability and hence eases the maintenance of the program.

5.2.3. Flexibility of Model

Modelling involves analysis and representation of a real world system. Persistent relationships between objects and interaction between objects are two kinds of behaviour we may often observe. For instance, consider the following scenario. "A table lamp lights up when the switch is at position ON and it turns off otherwise." – this is an persistent relationship. There is interaction between a man and the light switch so as to change the state of the switch. This interaction does not change the persistent relationship between the brightness of the table lamp and the light switch, but some interactions do. A sudden impact on the table lamp may cause breakage of the filament so that the relationship is changed to "the table lamp will not glow irrespective

of the position of the light switch". This shows that a persistent relationship is not necessarily permanent; it is subject to change by interaction.

We have already experienced problems, such as verification and concurrency, with imperative programming which disregards the persistent relationships; we have also the experience of using functional programming which stresses the permanency of persistent relationships – making use of higher-order function to prevent change of relations adds a degree of complexity to the relationships. Definitive programming paradigm enables us to describe persistent relationships without ruling out the possibility of relationship changes by interaction. Hence it is desirable for modelling.

Furthermore, a set of definitions shows not only the design of the model of current state, it also provides hints for change of design. The intelligent use of constants and formulae in defining variables indicates the flexibility of the model. Using the Jugs example as an illustration, the point *base* is defined in Figure 5.1 to be {1.c, n.r} where *n* is currently defined as 20. Redefining *base* as {1.c, 20.r} rather than {1.c, n.r} does not affect the value of the point *base* and hence the whole picture remains unchanged. But the definition

$$\text{base} = \{1.c, n.r\}$$

gives *base* a degree of freedom – the point *base* can be moved vertically without changing its definition but only changing the explicit value of *n*. Of course there is no rule to guarantee that the definition of *base* is fixed or that the definition of *n* is going to be altered, but the use of implicit formulae and explicit values in definitions suggests that the variables defined by explicit values are more liable to change and the variables defined by implicit formulae are more persistent.

Therefore, variables in a definitive notation are more than variables containing pure values; the formulae defining the variables are significant. In fact, they are more significant than the values. This is because the variables must specify a unique set of

values if sufficient definitions are given, but if some definitions are missing (i.e. the model is incomplete) the formulae define latent values of the variables.

5.2.4. Separation of Control and Presentation

Since definitive notations are special purpose notations, a script written in a single definitive notation is generally insufficient for specifying the whole application. On the other hand, the usefulness of definitive notations is not undermined by this; a script can still be used to model a particular aspect, such as the screen layout, of the application.

With reference to the Jugs example, the Scout definitions only describe the screen layout. They do not specify how the variables like contA and valid1 are maintained. In fact the control in the Jugs example is written in EDEN, a general purpose definitive language. A way of integrating definitive notations via EDEN will be discussed in the next chapter. The basic idea is to translate different kinds of definitions into a single definitive language so that variables of different definitive notations can communicate via definitions. This means, for example, that in order to animate the Jugs layout, designed in Scout, it is only necessary to append the EDEN script and a set of actions that defines the Jugs control. Therefore, a definitive paradigm for representation of state provides a neat way of separating control and presentation. The advantages of the separation are:

- The development of the control can be made independent of the development of the presentation; this leads to faster program development and aids the division of labour.

- Different views of the same application are possible at the same time. For instance in the Jugs example, we can execute the Scout display specification together with the display specification, suitable for a TTY display, that is incorporated in the

original EDEN Jugs control[2]. As a result, another Jugs display will appear on a TTY terminal.

5.3. Exploratory Screen Layout Development

The screen layout target is not always known at an early stage of screen layout design. A practical way of screen layout design is to obtain a first approximation and then gradually evolve the design through prototyping and experimentation. During an exploration of design, one of the following activities may be performed:

1. Removing unwanted items

   Example: In our early Jugs program, instead of the 5th option – pour water from one jug to the other – we had an option for pouring water from Jug A to Jug B and another option for pouring from Jug B to Jug A. Although in the actual menu-driven simulation the two menu options for pouring are redundant, the full range of menu options is useful for general simulation of pouring. On this basis, it is not clear whether we should have one menu option for pouring or two. But when we decided to accept the single menu option, options 5 and 6 were then removed.

2. Displaying new items

   Example: Following the example above, after the deletion of the two 'pour' menu options, the current option 5 was added.

3. Relocating the display items

   Example: Changing base so that the whole display shifts. Several tests may be necessary because where base should be is subjective.

---

[2] Written by Dr Meurig Beynon. See Appendix D.

4. Modifying relationships between variables

   Example: The message box may be relocated so that it lies below the menus. This action will break the geometrical relationship between the location of the message box and the capacity of Jug B (see Figure 5.2) and establish a new relationship between the message box and the menus.

5. Testing of design – changing the parameters or testing data

   Example: Changing contA and contB to see if the menus and the message box behave as they are intended.

5.3.1. Convenient State Changes

Although redefining a variable may cause changes to the values of many variables and hence the screen display, the only difference the redefinition makes to the definitive state is the definition of that particular variable. Therefore, reversing the changes made by the redefinition only requires restoring the original definition of the variable. Thimbleby argues that the user of an interactive system must be able to undo errors. With a good undo available, users will be encouraged to experiment with the system [Thimbleby90]. In our current system, no undo facility has been implemented. It is our intention to leave the system in a raw operational mode so that there is no fancy user interface to distract our attention from developing higher level control for transitions of definitive states. However, the simplicity of undoing the effect of a definition is an advantage of definitive notations for exploratory design.

5.3.2. Flexible Definition Arrangement

Changing the two pour menu options to one pour option in the jugs example involves replacing of the definition:

display screen = (... / pourAtoBwindow / pourBtoAwindow / ...);

by the definition:

display screen = (... / pourwindow / ...);

with the addition of the following definitions:

```
window   pourwindow = {
                frame:(menu5box);          string: pupilmenu5;
                bgcolour:background5;      fgcolour:foreground5
         };
string   pupilmenu5 = "5:Pour";
string   foreground5 = if valid5 then "black" else "white" endif;
string   background5 = if valid5 then "white" else "black" endif;
box      menu5box = [menu4box.ne + {1.c, 0}, 1, strlen(pupilmenu5)];
```

Listing 5.3: The Scout Definitions Relating the Pour Menu Option

Listing 5.3 defines all the necessary information required to display what can be seen on the screen as the "Pour" menu option (i.e. the *region*, *content* and *attributes* of the window are all defined). The only piece of missing information is menu4box, which is part of the display information of another menu option. Listing 5.3 is therefore similar to a window object in object-oriented programming terms, except that in our paradigm no information hiding is assumed. This grouping of definitions here and the grouping of definitions illustrated in Listing 5.1 and 5.2 shows two grouping methods with different emphasis. One groups the definitions relating a visible window whilst the other groups the definitions according to their functionality. Flexibility of definition arrangement is possible because the ordering of definitions in a script is insignificant. The advantages of having this flexibility are:

1. One can develop a script in whatever way is most convenient to the current stage of development. Perhaps in the beginning the Scout display is developed in phases such as specifying regions, specifying contents and combining them to form a

screen. Later, exploratory design is benefited by developing the screen window by window.

2. Regrouping of definitions will not affect the meaning of the script. It is possible therefore to develop tools to rearrange definitions in ways that can assist our understanding of the script. Particularly useful arrangements might be obtained by sorting the definitions by types or by their dependency hierarchy.

5.3.3. Design and Simulation Joined Together

In the definitive paradigm, there are many types of activities but only one type of operation – redefinition. A redefinition may produce both the effect of i) changing the model and ii) testing (or simulating) the model. For examples, redefining widthA changes the layout design but redefining contA is part of the simulation process. This shows that definitive programming encapsulates design and simulation in the same process; when the programmer is satisfied with the design, a program is ready for use [CW89].

Is it a good idea to merge design and simulation? In other words, should the user be allowed to exert such power to change the design of a program? Although there is no distinction between data and program in conventional computer architecture, a program in a conventional high-level programming languages is not usually changed during its execution.

The simulation referred here is part of the software development process. In a software developer's role, one has the right to modify one's software. But to a user, the development of the software is supposedly frozen. Only certain ways of interaction to the script is expected. Therefore, it is more appropriate to ask whether there are any convenient ways of restricting the user's power to modify a definitive script.

There is a danger in this discussion of giving the reader an impression that definitive script is a program. Since a set of definitions is meant to model a state but

not to handle inputs and the transitions to the state according to the inputs, we should not generally treat a definitive script as a program, at least not a complete program. However, since a definitive state may contain complex relationship between variables, a change in the value of a variable may induce a large amount of value changes to other variables and to the output. For some applications where dynamic change of relationship between variables is not required, for example the vehicle cruise control simulation example in Chapter 7[3], simulating the application is essentially changing the input parameters in the conventional programming sense. For this kind of application, a definitive script may be considered as a program with a different input specification. While the expected program may accept a number entered in a particular dialog box, the definitive script accepts a redefinition of a variable to that number.

In order to turn a definitive script into a program, an interface transforming the user input into redefinitions is needed. A few possibilities are explored in this thesis:

1. Extend definitive notations to a complete language with transition control. One such language is LSD; this will be discussed in Chapter 8.

2. Write a simple interface program which fulfils the required input specification and generates appropriate redefinitions for the definition evaluator. Some software tools such as tooltool [Musciano88] exist to assist the creation of this interface. The current Scout system has also been extended in such a way that user-input can be captured and transformed into required definitions. Section 7.3 will explain the mechanism in detail.

3. Transform the definitive script into a conventional language, where the transformed program only allows changes to certain variables. Since conventional programming

---

[3] There are changes of variable relationships in the Jugs example. When a Pour menu option is selected, a relationship between the water levels of the two jugs will be established so that the reduction of water level in one jug will simultaneously raise the water level of the other. But after the pouring action is finished, the relationship will no longer exist.

separates the development of a program from its simulation, the transformation effectively freezes the development of the program. The rest of the chapter will discuss this transformation process.

Attempts at transforming a definitive script into an imperative program were made by Michael [Michael89] and Hui [Hui90]. Trident is a software tool designed to convert an EDEN program into C program. EDEN and C are chosen because they have a large common subset of data types and operators.

```
/*declare          /* X and target are the variables to be changed */
change(X, target);
*/

X = 0;             /* this redundant assignment conveys type information */
target = 36;
sqrx is X * X;
correct is sqrx == target;

proc problem : target {      /* action for visualising target */
    writeln("X is the square-root of ", target, ". What is X?);
}

proc result : correct {      /* action for visualising correct */
    if (correct)
        writeln("You've got it");
    else
        writeln("Then square of ",X," is ",sqrx," not ",target);
}
```

Listing 5.4: Square-root Guessing Program in EDEN

Listing 5.4 is a simple EDEN program for guessing the square-root of a given number. The definitions of X, target, sqrx and correct form a definitive state. The two actions are used for visualising the definitive state; they serve the same function as Scout definitions. On redefining the variable X, a message stating whether or not X is the square-root of the target, initially 36, will be displayed; when the variable target is redefined, a message stating the new goal of the problem will be displayed. In EDEN, a richer range of interaction is allowed (for example change the problem to solving

cube-root instead of square-root), but as indicated in the first three lines of EDEN comments, only X and target are the intended input of the program[4].

```
int correct, sqrx, target, X;

_user_input() {
        char name[80];
        while (!feof(stdin)) {
                scanf("%s", name);
                if (!strcmp(name, "X")) _X();
                if (!strcmp(name, "target")) _target();
        }
}

_target() {
        scanf("%d", &target);
        problem();
        correct = sqrx == target;
        result();
}

_X() {
        scanf("%d", &X);
        sqrx = X * X;
        correct = sqrx == target;
        result();
}

result() {
        sqrx = X * X;
        correct = sqrx == target;
        if (correct) {
                printf("%s\n", "You've got it");
        } else {
                printf("%s%d%s%d%s%d\n",
                        "The square of ", X, " is ", sqrx, " not ", target);
        }
}

problem() {
        printf("%s%d%s\n", "X is the square-root of ", target, ". What is X?");
}

main() {
        X = 0;
        target = 36;
        problem();
        result();
        _user_input();
}
```

Listing 5.5: Translated Square-Root Guessing Program in C

---

[4] To the EDEN interpreter, these few lines are comments. They are ignored by the EDEN interpreter but they are introduced to give essential information to the Trident translator.

Given the definitive state, the dependency between the variables can be calculated. Therefore, having specified which variables are subject to change (X and target in this case), the Trident translator can generate procedures to emulate the effect of redefining those variables in EDEN. In the procedural program constructed by the Trident translator constructs, each definitive variable is emulated by an imperative variable. The value of such a variable is maintained by repeated re-assignments. These assignments ensure that all the variables essential for calculating the formula associated with that variable are evaluated before the variable is assigned the value of the formula. Listing 5.5 is the transformed C program.

In its present state, Trident is a highly restricted translator. The generated program has a restricted form of input: it can only accept input of the form:

```
variable-name value
```

This is usually not the required input format. A better version of Trident should allow the specification of the expected input format.

Despite the fact that the input format is restrictive and that the current Trident translator is only able to translate very small examples, there is still an essential difference between the translated program and a 'normal' guessing program. Normally, a user cannot alter the target until the correct answer is given and he is not supposed to keep on changing X when he has achieved the correct answer. At some stage, a guessing program would usually provide a channel for exit. Because the EDEN program in Listing 5.3 allows X and target to be redefined at any stage and never terminates, the translated program also inherited these properties. It would not be difficult to enhance the Trident translator to generate a more appropriate program. The change construct informs the translator what the user is privileged to act upon in a definitive state. It is not hard to imagine a version of Trident translator which can grant conditional privileges. The user might start with the privilege to change X; if correct

becomes true, the user might be privileged to change target; termination of the program means that the user has no longer any privilege to change the definitive state.

In the discussion on Trident above, we are in effect exploring a non-traditional way of software development. It is not writing a higher level program satisfying the specification, then translating it into a program in the target language, as in the case of writing a C++ program and then translating it into a C program for execution. It is first writing a program which has a different input specification (a specification that allows a wider range of input, and where the input formats are also different), then developing a program satisfying the original specification by restricting the range of the input and converting their formats back to the original specification. The situation can be depicted in Figure 5.4, where a larger area means a higher degree of freedom in implementation.

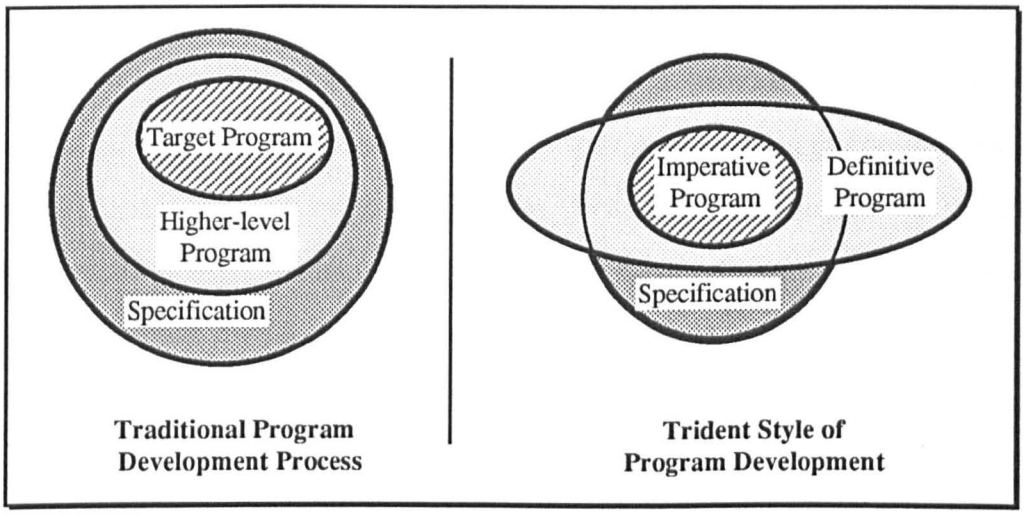


Figure 5.4: The Trident Way of Software Development

During the development of a definitive script, the design and simulation processes are interleaved. This shortens the editing stage of the exploratory software development cycle. Trident shows that it is possible to freeze the development of a definitive script and use the script to develop an executable program. In this way, the final program has a better run-time performance.

5.4. Summary

Writing a script of definitions and performing on-line modification of the script is the simplest way of using definitive notations. The main use of these interactions is to develop a definitive script which models the state or part of the state of a system. The on-line modification facility favours exploratory development of the definitive state model. There are many more factors of definitive notations that are advantages for exploratory development of definitive state. To help the programmer to comprehend the state, definitive notations have domain-specific data types and operators. The way of expressing dependency in a definition provides a strong but modifiable link between variables. This allows a neat separation of internal state and its presentation. It is particularly helpful in visualising abstract information such as the speed of a vehicle. Also tools may be built to rearrange the definitions to provide useful insight for the programmer. To help in the editing phase of exploratory design, definitions have potential for building up a good undo facility. In addition, redefinition can serve both to effect redesign and simulation. This means that the development of a definitive state can be done in a continuously executing environment.

6

Integrating Definitive Notations

Although a simple definitive notation may be good at addressing a particular aspect of an application, a stand-alone definitive notation has limited usage. Dealing with a large problem often requires the cooperation of several definitive notations. This chapter describes and evaluates our effort to integrate several definitive notations into a single system. The essence of the integration is the design and implementation of the Scout notation so that pictures described by other definitive notations can be combined to form a screen display. For this reason, we have called our effort at integrating definitive notations *the Scout Project.*

6.1. Motivation for the Scout Project

A motivating example of visualisation in mathematical research is considered. This example is based upon [BYAB91], where the mathematical context is discussed in

greater detail. In particular, the example illustrates the visualisation of combinatorial structures associated with arrangements of four lines.

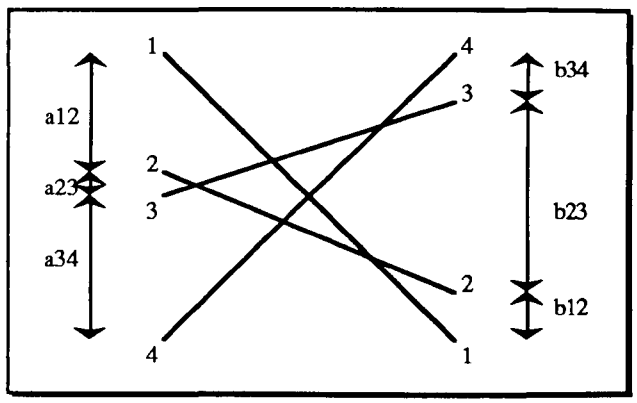


Figure 6.1: An Arrangement of Four Lines

In Figure 6.1, the line arrangement represents a sequence of transpositions of the permutation 1234 to 4321. The sequence can be obtained by interpreting the crossings of the top, the middle and the bottom pair of lines as transpositions of the first, the second and the third line pairs. In this case, on scanning the arrangement from left to right, the crossing sequence is then 213231. The crossing sequence corresponds to a shortest path, or a *geodesic*, in the Cayley diagram for the symmetric group S_4 as depicted in Figure 6.3. Two geodesics that differ only in the order of disjoint transpositions, such as 213231 and 231213, are equivalent. A *poset* to represent its equivalence class can be derived from a geodesic. Figure 6.2 is the poset representing the equivalence class of the geodesic 213231.

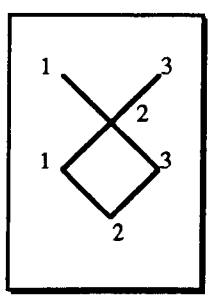


Figure 6.2: A Poset Representing the Line Arrangement in Figure 6.1

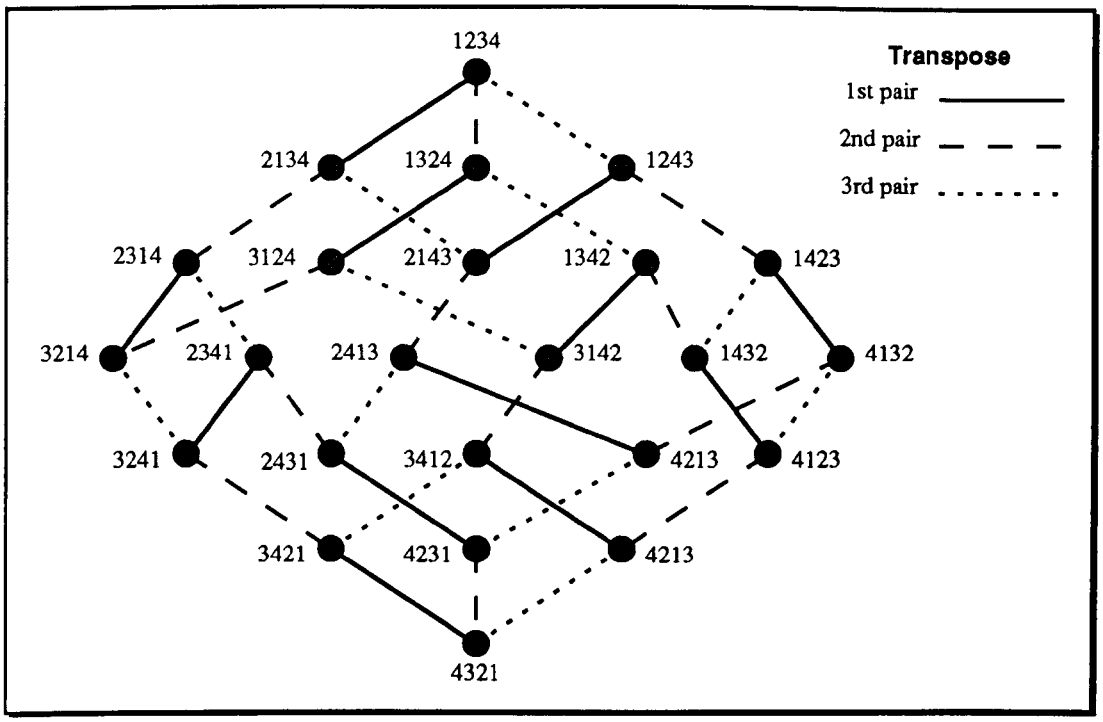


Figure 6.3: An S_4 Graph

A typical mathematical research activity is to explore the relationships between the line arrangements, the posets and the geodesics by varying the ratios a12:a23:a34 and b12:b23:b34. Therefore, related figures similar to Figure 6.1, 6.2 and 6.3 have to be displayed at the same time. This what-if exercise is conveniently dealt with by definitive notations. However, this visualisation problem does pose two challenges concerning the use of definitive notations.

The first challenge is the choice of definitive notations to describe the figures. We have DoNaLD, a definitive notation for line drawing, which suits the display and manipulation of the line arrangements. We also have ARCA, a definitive notation for displaying combinatorial diagrams, which is most suitable for visualising posets and Cayley diagrams such as S_4. But when we think of how to transform a poset into a group of geodesics of displayable form (such as the string "<213231, 231213>"), we need more powerful and more general functions and data types than those available in both DoNaLD and ARCA. In the last chapter, we explained that one of the advantages of definitive notations is their being specific in design. We cannot, therefore, expect to

build a definitive notation which has general data types and operators and, at the same time, includes all the special data types and operators of DoNaLD and ARCA. Realistically, variables in one definitive notation should be able to reference those of another definitive notation. Two conclusions can be drawn:

1) In the definitive paradigm, the way one variable relates to another is expressed by means of a definition of the variable written in terms of the other. A variable in one notation should, therefore, relate to a variable in another notation by means of definition as well.

2) Because we need to set up references across definitive notations, and these linkages are fundamentally definitions involving variables from different definitive notations, all the variables should be put into a common definition store. This implies that the definitive notations should be implemented in such a way that all the definitions will be translated into a common form.

The second challenge presented by the visualisation problem is the need for organising the display. DoNaLD and ARCA at their early stages considered only the display of line drawings or combinatorial diagrams in isolation. How these pictures related to each other and the wider considerations for integrated use in an application were ignored. This means that both DoNaLD and ARCA generate independent pictures on the screen. But in this context, several pictures are going to be displayed in the same application. Comments and labels are also needed to identify and explain them. Another definitive notation for describing screen layout is essential.

In addition to these two challenges, a definitive notation for general mathematical computation and string manipulation is obviously needed to complement other special-purpose definitive notations. EDEN can be this notation provided it is used in a disciplined way. This is because EDEN has imperative features as well as definitions.

There have been no previous attempts to integrate definitive notations. The definitive notations that are currently available have not been designed and implemented with integration in mind. The aim of the Scout Project is to build bridges between definitive notations so that they can be evaluated harmoniously within a single system.

6.2. Scope of the Scout Project

The scope of the Scout Project is:

1) to design and implement a definitive notation for describing screen layout – Scout;

2) to modify the implementations of the current definitive notations so that they can be evaluated together;

3) to provide guidelines for future development of definitive notations.

The design and implementation of the Scout notation has already been discussed in Chapter 4. The rest of the chapter will concentrate on the general framework and other guidelines for implementing definitive notations.

6.3. Implementation of Definitive Notations

Spreadsheets give useful insight into the definitive paradigm. In a spreadsheet, each cell is a variable of type string or number. By defining a cell with a formula, the system will automatically recalculate the formula whenever any cell it depends on is changed. The up-to-date value of the cell is then displayed in the cell. We can imagine that there are implicit actions which update the values of the variables on the screen. Similarly, there are such implicit actions in the implementations of definitive notations. For example, each graphical object in DoNaLD has a representation on the screen. For a point variable there is a dot to represent it; for a line variable, there is a linear set of points and so on. So there will be a plot_point action in the implementation of DoNaLD

to be called automatically when a point variable is updated, and likewise a plot_line action for a line variable.

Section 6.1 indicates that a general-purpose definitive language is required so that any definitive notation can be translated into it. Besides, as we have just explained, this language should also allow user-defined actions to be defined for displaying variables of different data types. A definitive language satisfying both requirements is EDEN (cf. [Yung89]).

6.3.1. Steps for Implementing a Definitive Notation

The following steps must be taken for implementing a definitive notation in EDEN. This method has been used for implementing Scout and DoNaLD. Among the two notations, the implementation of DoNaLD will illustrate the implementation method more clearly. Therefore, the examples associated with each step are all taken from the implementation of the DoNaLD notation.

1. Derive a scheme for translating variable names into EDEN variable names. For example:

DoNaLD name	EDEN name
table	_table
table/drawer	_table_drawer
table/drawer/width	_table_drawer_width

2. Emulate the data types and operators using EDEN data types and user-defined functions. Almost inevitably this will make use of the list structure in EDEN because list is the only complex data type in EDEN. For example:

DoNaLD type	EDEN type
integer	integer
point	['C', integer, integer]
line	['L', point, point]

DoNaLD operator	EDEN operator/function
div	/

+ (vector sum)	func vector_add {
	para p1, p2;
	return ['C', p1[1] + p2[1], p1[2] + p2[2]];
	}

3. The underlying algebra of the target notation has been implemented through steps 1 and 2. To complete the implementation, the required implicit actions are emulated using EDEN's user-defined actions. For example:

DoNaLD code	EDEN action specification
integer i	No action[1]
point p	proc P_p: _p { plot_point(&_p); }[2]
line L	proc P_L: _L { plot_line(&_L); }

4. Write a preprocessor to translate scripts in the definitive notation into EDEN in the way implicitly defined by steps 1 to 3.

6.3.2. Run-Time Structure

The run-time structure of the implementation of a typical definitive notation such as is described in §6.3.1 is depicted in Figure 6.4. The library contains the EDEN implementation of the underlying algebra together with the functions and procedures useful for the EDEN actions.

---

[1] No action required because integer variables do not have any graphical representation in DoNaLD.

[2] This & operator is similar to that in the C language, it returns the address of the variable. Plot_point and plot_line are EDEN (user-defined) procedures which do the plotting.

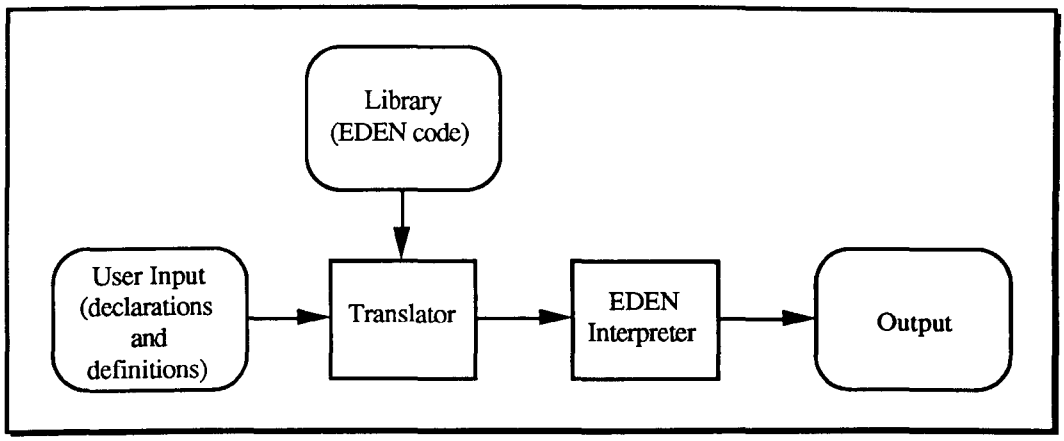


Figure 6.4: Run-time Structure of a Definitive System

The integrated Scout system is implemented in this fashion. Figure 6.5 shows the run-time structure of the Scout system.

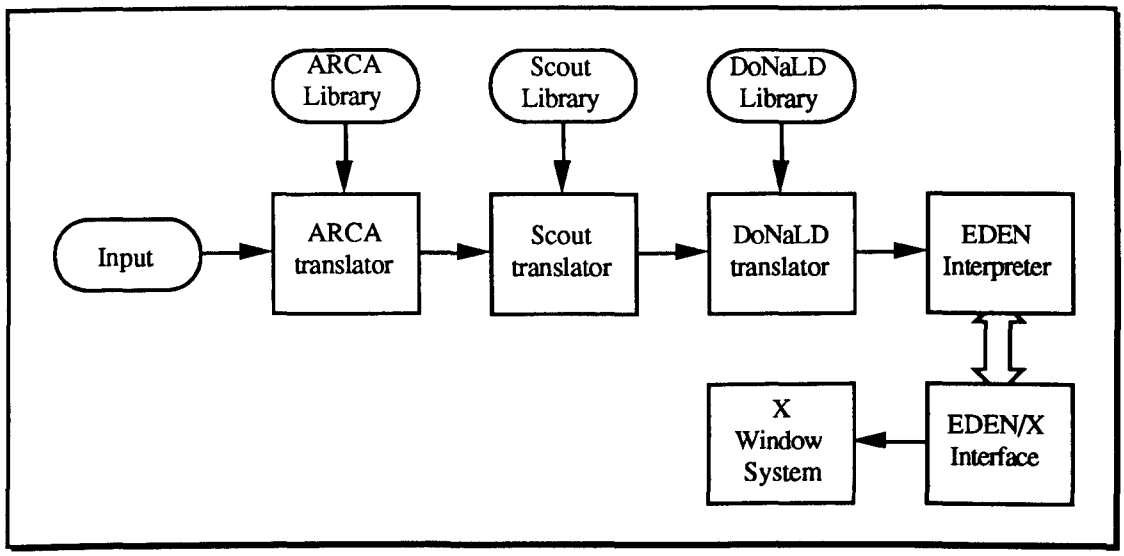


Figure 6.5: Run-time Structure of the Scout System

The Scout system is implemented in a UNIX environment. The user's input is pipelined through a series of ARCA, Scout and DoNaLD filters (the ordering of the filters is not important) which translate ARCA, Scout and DoNaLD definitions into EDEN definitions. These definitions together with the functions and actions in the libraries are interpreted by the EDEN interpreter. Graphical outputs are generated by an EDEN/X interface which is actually an X Window client. When the graphics display needs to be updated (i.e. some EDEN actions generate graphics output), the EDEN

interpreter will interact with the EDEN/X interface, which in turn will interact with the X Window server to produce graphics images.

6.3.3. The Choice of Graphics Interface

In the first implemented definitive notation – ARCA, output was generated via a plotter. After EDEN had been developed, more definitive notations were prototyped. At that time, only the SunView window system [Sun84] was available and hence DoNaLD, ARCA, CADNO and the preliminary version of Scout were developed on SunView. While other definitive notations used the SunCore window library for graphics display, the original Scout used the WW library from Rutherford Appleton Laboratory [Martin87] to simplify the implementation. At that stage, interfacing a graphical library required the inclusion of a large number of graphics library functions and routines as part of the EDEN built-in functions. This meant that a few versions of EDEN were created, each customized for one particular graphics library. Moreover, the definitive notation implementation libraries had to be developed for each version of EDEN. All these made maintenance of the system difficult. Later, two more powerful network window systems became available – the NeWS window system [Sun87] and the X Window system [GO90, MAS88]. In view of the coming thrust of more powerful and standard window systems, changing the window environment was unavoidable. Instead of creating yet another version of EDEN, a more flexible scheme for interfacing with the window system was derived – a graphics interface was separately built. Such separation is a standard issue in user interface management systems [HH89, Foley87, Gray89] and it has a number of advantages:

1. The graphics interface and the EDEN Interpreter can be run in parallel.

2. Other graphics interfaces may be built without modification or enhancement to the EDEN language or other definitive notation implementation library.

3. The graphics interface is reusable by other systems, not necessarily definitive systems.

The X Window system was chosen for our current implementation in preference to NeWS. This is because NeWS is written in PostScript whereas our prototypes were all written in the C language and, more importantly, NeWS was a buggy system. The choice proved wise because X Window is now more popular than any other window system. Currently, our system is supporting X version 11 release 5.

The EDEN/X interface program, called EX, was developed by me. It is linked with EDEN via a *message queue*. *Message queue* is a UNIX System V inter-process communication channel. It allows multi-directional communication between a number of processes. EX accepts a simple command language for creating windows, drawing graphics primitives, displaying text and processing queries such as the size of font in use. So long as this command language does not change, no alteration to the implementations of the definitive notations is needed even if the implementation of EX changes.

6.4. Other Guidelines for the Design and Implementation of Definitive Notations

6.4.1. Bridging Definition

As suggested in §6.1, bridging definitions provide a way of communicating between definitive notations. For example, we can use Scout to specify textual annotations of a DoNaLD picture. Listing 6.1[3] is an example of this kind. The information shown in the window doorButton is "Close Door" or "Open Door" dependent on the value of the

[3] Extract from Figure 1 of [BY90].

DoNaLD variable door/open. The interface between the Scout and DoNaLD notations is a bridging definition[4]:

integer DoorIsOpen = **DONALD boolean** door/open;

which, when translated into EDEN, becomes:

DoorIsOpen **is** _door_open;

In general, because the implementation of the Scout data types may be different from the implementation of the corresponding DoNaLD data types, type conversion is needed when the bridging definitions are translated into EDEN definitions. So the translated bridging definitions normally take the form:

Scout-variable-name **is** *convertor(DoNaLD-variable-name)*;

Similarly, bridging definitions can be added to DoNaLD and other notations so that a complicated communication network can be achieved. Bridging definitions are in concept no different from other definitions, and the general rule of non-circularity still applies to them. Care should be taken when writing bridging definitions so as not to violate the rule.

```
integer DoorIsOpen = DONALD boolean door/open;          # a bridging definition
point miscMenuRef = {250, 400};
point doorButtonPos = miscMenuRef + {strlen(plugMenu).c/2, 1.r};
window doorButton = {
        frame:   ([doorButtonPos, 1, strlen(doorMenu)]),
        string:  doorMenu,
        border:  1
};
string doorMenu = if DoorIsOpen then "Close Door" else "Open Door" endif;
```

Listing 6.1: Program Fragment of the Scout Notation

[4] Not implemented yet. In the actual listing, the EDEN definition "DoorIsOpen is _door_open;" is written instead.

6.4.2. Naming Scheme

For historical reasons, each definitive notation is translated into EDEN according to its own naming scheme. For instance, neither Scout nor ARCA changes the variable names during translation, which means that a Scout integer i, an ARCA integer i and an EDEN integer i will all be cast into the same EDEN variable. Name collision may happen to the variable names, function names and action names of the definitive notation implementation libraries. Without precise guidelines, function names such as *int_mult* (integer multiplication) may be unconsciously chosen for the implementation of two similar but distinct functions of two definitive notations. Therefore, some precautionary steps or guidelines on the naming are necessary to supplement the implementation steps listed in Section 6.2.

A possible suggestion is that every variable name begins with characters identifying which definitive notation it belongs to. For example SC_i, AR_i and DO_i may be the translated names of the variable i in Scout, ARCA and DoNaLD respectively.

6.4.3. Switching Between Notations

In order to incorporate several definitive notations into a single system, the Scout system uses a method similar to the way the preprocessors of `roff` (the standard UNIX text formatting language) works. A block of definitions of a definitive notation is preceded by a declaration of the notation name. For example:

```
%scout
...
Scout definitions
...
%donald
...
DoNaLD definitions
...
```

The individual translator will translate only the lines from the notation declaration downwards until the declaration of another definitive notation is reached; the remaining lines are untouched. In this method, all the translators are running independent of one another. This provides flexibility for future extension to the system.

6.4.4. Other Considerations

Comments are useful information for maintaining programs. When there is only one notation within a system, how comments are inserted is not important. But when a system incorporates several notations, it is logical to use only one format of comment for the whole system rather than have different formats for different notations. For the same reason, there should be a consensus on the way definitions are separated from one another. Ideally, these considerations should influence the design of definitive notations.

There is a historical problem in that the design and implementation of our definitive notations preceded the integration plan. There has been no consensus on the format of comments and definition separator. A DoNaLD or Scout comment starts with a # sign till the end of the line; EDEN comments are enclosed by a /* ... */ pair; EDEN and Scout terminate a definition with a semi-colon but there is none for DoNaLD or ARCA. Since the # sign is a postfix operator in EDEN, it may be confusing if the # sign is set as an indicator for the start of comment in the unified system. This shows that in choosing the format of comments we should consider whether the comment sign is likely to bear any other meanings in other, may be even future, definitive notations. Since EDEN is the core notation in our system, it is most reasonable to preserve its

style. Another good reason for adapting this convention is that the EDEN comment sign consists of a combination of two symbols, rather than a single symbol that is likely to be used as an operator. Concerning the definition separator, it is usually easier to parse a definitive notation with a definition separator, and EDEN has one. Therefore, we recommend that the /* ... */ pair and ';' should be the standard comment sign and definition separator for all definitive notations.

6.5. Evaluation of the Scout Project

This section describes the current status of the Scout System, and summarises our experience of its design, implementation and application.

6.5.1. Modelling, Understandability and Usability

ARCA, DoNaLD, CADNO and EDEN were originally designed and implemented by different people to run in different environments. With the development of the Scout notation, it becomes possible to bring several definitive systems together in the Scout environment. Currently, the Scout system can incorporate ARCA, DoNaLD, EDEN and Scout notations. More definitive notations can be integrated into the Scout system as time goes by.

The ability to integrate definitive notations allows the breaking down of a large programming task into sub-tasks. Each task may be effectively developed under a suitable definitive notation. Scripts of definitions can simply be put together to form a required program. The only slight modification may be the addition of bridging definitions.

Special-purpose notations can improve software development productivity [BCMZ88, Ramsay87]. They allow the programmers to work in a higher level language closer to the problem domain. The programs created are easier to understand and debug. In exploratory software design especially, shortening the understanding stage is absolutely crucial.

One drawback of using a higher level, domain-specific notation is that some efficiency is lost in execution. Although we may be prepared to sacrifice a certain amount of run-time speed for the sake of quicker development, greater robustness and reliability, run-time efficiency is also a major concern in exploratory program development. This might be used as an argument against developing definitive notations instead of perhaps developing programs in EDEN directly. However, it can be argued that the time efficiency is an insignificant loss. The evaluation of a definitive notation consumes time in two areas: the translation process and the evaluation of EDEN definitions. Since the source notation and the target language (EDEN) are both definitive, the translation time is short. Also, because the principal difference between the original script and the translated one is the change of underlying algebra, the translated script should have the same order of efficiency as if the problem is directly solved in EDEN.

The designers of multi-paradigm programming languages such as LOOPS and NIAL need to consider the pros and cons of practising mixed programming styles [Bobrow85, Smillie88, KKW87, MGJ84]. Similarly, we have to justify the integration of several notations in relation to the issue of usability: Is it realistic to learn so many notations and then use them? Part of our justification is a familiar argument that is advanced for mixing programming styles: improved expressiveness can be bought at the cost of introducing diversity of notation. Our approach has some advantage over mixing programming styles since all the notations in our integrated system are based on the same definitive programming framework. Learning a new definitive notation is like learning new vocabulary whilst learning a new style of programming is like learning new grammatical rules. Therefore, the learning time for definitive notations should be reasonably short. Moreover, the definitive notations are application-oriented. The new vocabulary to be learnt should be familiar jargon that is easily picked up by the potential users. For this reason, it should be easy for someone familiar with the intended application to learn the underlying algebra of any particular

definitive notation. It is, however, reasonable to expect the subsidiary parts of the notations, such as comments and definitions separator, to conform to the same convention.

One aim of developing definitive systems is to assist exploratory software design by applying definitive principles to modelling the state of the real world (i.e. using definitions to capture the relationship between objects). Through observations and experiments by redefinitions, the problem under consideration can be more easily investigated. By integrating definitive notations, different aspects of a problem can be modelled in the most appropriate notations and the scripts can be efficiently combined. Hence, the effectiveness of definitive state modelling is maximized.

6.5.2. Support for Iterative Design

The visualisation problem described in §6.1 has been programmed using the Scout system. The script and its output are shown in Appendix E. Note that the organisation of the script reflects the manner of its development; it is not well organised. You can find, for examples, a fragment of DoNaLD definitions in the midst of some ARCA definitions; the declarations of the variables are not grouped together; definitions of the same notation are not grouped together. The haphazard form of the script emphasises the suitability of the definitive system for interactive use in program development. Since only the last definition of each variable is significant, the ordering of the definitions is irrelevant and a programmer need not write the script in a particular order. One simple example found in the visualisation example is that the scaling factor, m, in the specification of poset was 50 at one stage. After observing the temporary output on the screen, it was set to 100 to get a better picture.

6.5.3. User interface

When switching from one definitive notation to another, the name of the new input notation has to be declared (see §6.4.3.). Whilst developing or experimenting with a

definitive script, switching between definitive notations is common. Experience tells us that it is very easy to forget to switch to the new input notation before entering a definition. Although the system already provides different prompts for different definitive notations to remind the user which definitive notation it is expecting, the user tends to ignore the prompts.

A better solution would be to provide a multi-window user interface[5]. This interface would provide one window for each definitive notation, one for system messages and one for the script of definitions. There would then be no need to type in notation declarations (such as %donald) by hand. Instead, the user would place the mouse in the appropriate window before entering a definition. The interface would determine whether a switch between notations was needed, and if necessary make the switch automatically.

It is desirable to have a window for the overall script and separate windows for individual definitive notations. On the one hand, it is important to keep a log of what the user has done and the system's response. On the other hand, since different definitive notations are specialised in and responsible for modelling parts of a problem, it is desirable to group definitions in the same definitive notation together, so that the analysis of each aspect of the problem may be done without the distraction from interpolated definitions of other definitive notations. In the visualisation example, the definitions for the line arrangement in DoNaLD, for the S_4 graph in ARCA and for the screen layout in Scout address different concerns and can be better understood if they appear in different sections of the program.

---

[5] A previous attempt was made to improve the user interface of the DoNaLD system [Iu89]. However, that attempt focused on improving the programming environment for the DoNaLD notation rather than on the integration of definitive notations.

7

Generalising Single-Agent Definitive Systems

In the last three chapters, the Definitive State-Transition model has been applied in the simplest way – all transitions to a definitive state are directly manipulated by the user. We have called this kind of system a "single-agent definitive system". The user is the only agent who initiates transitions between definitive states. The role of the computer is simply to maintain the definitive state, i.e. to maintain the values of the variables to be consistent with their formulae.

The integration scheme described in the last chapter indicates that a definitive state can be specified using an extensible underlying algebra. If the existing definitive notations are not suitable for a particular application, we may consider designing and implementing a new definitive notation. For instance, although we can draw real-

valued function graphs using DoNaLD (an example can be found in [CB92]), it would be better to have a separate definitive notation for plotting graphs. Therefore, conceptually, although a definitive script describes only one state, this state can be very complex.

This discussion leads us to consider whether single-agent definitive programming can be an appropriate general-purpose programming paradigm. Functional programming has already demonstrated that it is possible to represent an executable program using a single script that superficially resembles a definitive script [Research87]. Therefore, we are tempted to ask: "what are the practical differences between a definitive script and a functional program?". In responding to the question, Admira – A Definitive MIRAnda – is prototyped to assist the comparison of definitive notation and the typical functional language Miranda.

By using "programming a stack-based integer desk calculator" as a simple case study in §7.1.3, we find that a definitive script cannot specify interaction satisfactorily. To develop a general-purpose definitive programming paradigm, we must introduce mechanisms for state transition that is independent of the user. This is what we mean by "generalising single-agent definitive programming".

In §7.2 and §7.3, we consider what supplementary information is required to relate single-agent interaction with a definitive script to general-purpose programming. The Trident software briefly introduced in Chapter 5 shows that the gap between a definitive script and an executable program can be very narrow. In §7.2, we consider some techniques for the indirect construction of definitions that require the computer to take on a more active role than maintenance of variable consistency. In §7.3, we describe an extension to the Scout system to allow interaction with a definitive script to be event-driven. The work of this chapter leads us to consider a more general way of describing transitions to complement a definitive script. This will be the subject of the

next chapter, where the *agent* is introduced as a formal concept for specifying state transitions.

7.1. Definitive Programming vs Functional Programming

7.1.1. Design and Implementation of Admira

There are times when we want to define functions and higher order functions. For example, the script for the visualisation of line arrangements described in the last chapter can be written down more neatly if there are sorting and counting functions. The EDEN (imperative) way of defining functions and procedures involves a different paradigm for state representation; improper use of the procedural elements of EDEN will introduce the traditional problems of procedural programming into the definitive paradigm. On the other hand, functional programming does not provide referential information which is significant in specifying state [Beynon89]. To clarify these issues and to understand more about the relationship between functional programming and definitive programming, we have carried out a project to prototype a definitive notation for functions. The definitive notation is called Admira – A Definitive MIRAnda. Admira allows interactive definition and redefinition of functions and on-line function value queries. All these are done in a definitive fashion[1].

For the purpose of comparison, the underlying algebra of Admira is deliberately chosen to be as close to Miranda's as possible. Admira accepts almost the same grammar as Miranda does. The discrepancy between the two grammars is minor; the reason for altering the grammar is to ease the implementation of the prototype.

---

[1] Some functional programming environments like SML [Harper86] and KRC [Turner81] do support interactive redefinition of functions, but in a significantly different manner. In these systems, functions are evaluated at their point of entry. A redefinition of a function will not affect other functions even though their definitions make use of the function being redefined.

7.1.1.1. What Is an Admira Definition?

Each variable in Admira is of the function type; an Admira definition is a definition of a function. Since Admira adopts the syntax of Miranda, an Admira definition is essentially the definition of a Miranda function. It is obvious that what is called an equation in Miranda [Research87] can be treated as an Admira definition. Some functions in Miranda consist of more than one part. For example, for functions defined by cases, for functions with where-clauses, and for functions defined by pattern matching, the functional definitions can be naturally divided into parts, some of which may themselves be function definitions. However, these parts collectively form an Admira definition. In these cases, the name of the function is considered to be the definitive variable to be defined, and the whole definition of the function is considered as a definition. To assist the determination of the end of a definition, Admira requests a full stop at the end of each definition.

7.1.1.2. Storage of Definitions

While checking the syntax of the definitions, Admira keeps a list of all the dependent functions. For example the following defines 5 definitions (square is not counted because it is local to the definition of variance):

```
variance X = mean(square(X)) - sqr(mean(X))
                  where
                        square []   = [];
                        square a:X = (sqr a):(square X);
                  end.
sqr X = X * X.
mean X = (summation X) / (nitems X).
summation []   = 0;
summation a:X = a + summation X.
nitems []   = 0;
nitems a:X = 1 + nitems X.
```

Listing 7.1: An Admira Script for Calculating Variance

The dependency of the functions can be summarised in the following table:

Function Name	Dependent Functions
variance	mean, sqr
sqr	
mean	summation, nitems
summation	
nitems	

When the definition is put in the definition store, the name of the function, the function definition and the list of dependent functions are recorded.

7.1.1.3. Evaluation of Expression

The implementation of Admira does not follow the pattern described in §6.3. Definitive scripts in other definitive notations are translated into EDEN and the EDEN interpreter evaluates the definitions. Because the grammars of Admira and Miranda are so close, it is much easier to use the Miranda interpreter – mira – as the evaluation engine of Admira. When an expression is going to be evaluated, Admira will identify which variables have to be evaluated. These variables are those free variables in the expression together with their dependent variables. The functional definitions of these variables are then collected into a Miranda script. This Miranda script is sent to mira to provide an environment for the evaluation of the required expression. Finally, the expression is sent to mira for evaluation and the result is displayed through mira.

The complete implementation of Admira and the grammar accepted by Admira can be found in Appendix B and C.

7.1.2. Recursive Definition and Circular Definition

Superficially, Admira simply integrates the editing, compilation and the evaluation processes of functional programming, but by analogy with the use of other definitive notations such as DoNaLD, we wish to view an Admira script as a representation of a

state. Just as a line in DoNaLD may represent the state of a door, we would like to interpret a functional value in an Admira script as representing the state of an object[2]. We can then change the state (as represented by a set of functional definitions) by defining a new variable (introducing a new functional definition) or redefining an existing variable (changing the definition of the variable associated with an existing function name).

The existence of the Admira prototype suggests that everything that functional languages can do can also be done in a definitive style. This false impression arises because Admira is not executing in exactly the same way as other definitive systems do – Admira does not check for circular dependency of variables. Even with circular dependency, Admira will try its best to evaluate the variables. For example, the script:

```
ones = 1 : ones.

f 0 = 0;
f n = (g n) + 1.

g n = f (n-1).
```

recursively defines a value ones and two functions f and g (f is an identity function and g(n) returns n-1). These values and functions can be evaluated by Admira but the definitions cannot pass the dependency checking scheme applied in other definitive notations.

One reason for avoiding such rigorous dependency checking is the difficulty of distinguishing between circular definitions and recursive definitions from their form alone. The above examples clearly show that recursive definition is a concise way of

---

[2] In practice, it has been proven difficult to make use of an Admira script to represent a system state. This may be connected with the fact that Miranda data types were designed with abstract computation in mind, and are not directly associated with observable attributes of real world objects.

defining a complex data value. Circular definition, which also has self-referencing, is an attempt to define a data value in terms of itself, which is incomprehensible[3].

It is certain that circular definitions are not acceptable in definitive programming. Since recursive definitions bear the same form as circular definitions, how we should interpret recursive definitions such that they are compatible with ordinary definitions?

Recursive functions in functional programming have to have an operational interpretation. For example, the definition "ones = 1 : ones" in Miranda defines ones with the value of a list of 1's, but the definition "ones = ones : 1" will cause a *black hole*. The interpretation of a functional program requires the understanding of the evaluation mechanism. On the other hand, definitive notation is aimed at defining relationships between data. The interpretation of a definitive script does not need to refer to the evaluation mechanism, and preferably should not. The relationship between DoNaLD and Scout can illustrate this. With Scout, writing a DoNaLD script is like creating 2-D drawing on a large worksheet, just like working with pen and paper. The drawing is then viewed through a Scout window. Recursive definitions and ordinary definitions are therefore, in principle, defining objects in two levels. Respectively they are an operational level and a real world level. Their relationship has a parallel in the context of numerical analysis.

During evaluation of a function using a computer, we recognise two types of errors – truncation error and round-off error [BF85]. The term "truncation error" generally refers to the error involved in using a truncated or finite summation to approximate the sum of an infinite series. In other words, it is an error due to the use of an approximation function of the actual function. The "round-off error" is the error

---

[3] The name of God in Hebrew literally means "I am who I am", which is analogous to a circular definition I = I. No one can fully comprehend God, nor circular definition.

that results from replacing a number with its floating-point form. The round-off error affects the results in two ways. Firstly, it affects the parameters supplied to the functions and secondly, it affects the accuracy of the calculations. Therefore, when calculating "y = f(x)", y will in fact be given the value F(X) where F is the approximation to the function f and X is the approximate value of x. Although y is not assigned the intended value, the relationship y = f(x) still holds in the conceptual level. Only when we consider the computational level do we need to know what F and X are.

Recursive definition and ordinary definition are therefore addressing the two different kinds of relationships during computation. Whilst ordinary definitions address relationships such as those between y, f and x, recursive definitions address relationships such as those between f and F, and between x and X. Recursive definitions could be a means of specifying the underlying algebra over which ordinary definitions are defined. We prefer to lay down a computer model in terms such as π and sin() which are the ideal real values and functions in the real world; an implementation, on the other hand, is allowed to deviate from the real values and functions. For instance, an electronic calculator has a π key, but internally the π may only be a ten-digit figure.

If we take this view, we should be able to divide definitions of an Admira script into two kinds. One kind is for modelling the relationship between data, and the other for specifying the implementation of the underlying algebra. Unfortunately, because all variables and operators are functions, it is hard to distinguish from their form which definitions are defining the underlying algebra and which are not. In other definitive notations, the problem may seem smaller. This is because the values of the variables in other definitive notations cannot be treated as operators. However, it is still difficult to differentiate recursive definitions of values in the underlying algebra and circular definitions of variables without consulting their roles in interpretation.

Definitions as part of the definitive state model are qualitatively different from other functional definitions of the implementation of underlying algebra. The mixing of the two in a script, as demonstrated in an Admira script, surely makes the script difficult to understand. If part of the implementation of the underlying algebra is likely to be publicly made known, for example if a definitive notation allows user-defined functions, it is advisable to specify these functions in a section dedicated for specifying implementation.

7.1.3. State and Interaction

The difference between definitive programming and functional programming is highlighted by considerations of interactive programming. In definitive programming, the way in which states and possible interactions with states are represented is strongly related to the way in which we choose to observe a system. Take observation of a jar of gas as an example. A physicist wishing to study the jar of gas at a microscopic level might model the state of the gas in terms of the properties of individual gas molecules. These include the mass, the positions and the velocity of the gas molecules. The molecules interact with each other through collision. A chemical engineer, on the other hand, might be interested in macroscopic properties of the state of the gas such as pressure and temperature. Associated with different ways of recording the state are different ways of describing interaction. For instance, the physicist may like to change the velocity of the gas molecules on the wall of the jar whilst the chemical engineer may like to change the temperature of the jar.

Some models of state are simpler than the others. For example, the pressure of the jar of gas can be related to temperature by a simple formula, but the motion of individual gas molecule is harder to calculate. However, we argue that the choice of a model for representing states and interactions is essentially determined by how we observe. We should ask, "what kind of state description is appropriate under the method of observation in use?" [BR92]. It is inappropriate to obtain the pressure of

gas by an optical measuring instrument. A barometer, on the other hand, cannot indicate the positions of individual gas molecules.

Traditional solutions to the problem of interactive programming in a functional paradigm ignore the issue of choosing an appropriate mode of observation. For example, consider the problem of writing a simple stack-based integer desk calculator in a functional style. The following is a solution taken from [Wadge85]. It is written in pLucid, a functional dataflow language.

```
hd(stack) whenever command eq "p"
   where
      stack = [] fby
            if isnumber(command) then command :: stack else
               case command of
                     "+": (op1 + op2) :: stack2;
                     "-": (op1 - op2) :: stack2;
                     "*": (op1 * op2) :: stack2;
                     "/": (op1 div op2) :: stack2;
                     "p": tl(stack);
                     default: error;
               end
            fi;
      op1 = hd(stack);
      op2 = hd(tl(stack));
      stack2 = tl(tl(stack));
   end
```

Listing 7.2: A Stack-based Integer Desk Calculator in pLucid

The stack program above accepts an input variable command which is a stream of stack operations and output a stream of values. Stack, as well as op1, op2 and stack2, are what Wadge has called *time-varying values*. If we attempt to output stack in the course of calculation, we notice that stack itself is a data stream. If we consider the functional program in Listing 7.2 as a description of a single state, the state under observation is a stream of stacks. This is not ideal because we normally observe a stack at a time rather than a stream of the stacks. It is, therefore, unnatural to write a definitive script that describes a 'stack' which is in fact a collection of many stacks. This example indicates that although we could avoid specifying state transitions in definitive programming by adopting a complex underlying algebra as in Miranda and

pLucid, it is preferable to preserve transitions so that the state described complies with our usual understanding of its real-world counterpart. This is the reasoning that leads to the introduction of agents in specifying state transition.

So far we have no satisfactory solution to specifying a stack within a single-agent definitive paradigm, and we are doubtful if there is such solution without specifying some sort of agent to initiate transitions of states. In the following sections, we will consider a number of proposed and existing programming techniques related to current definitive notations that may be viewed as *agent-like*.

7.2. Agent-like Abstractions and Definitive Scripts

7.2.1. Meta-Definition

In our current systems, there are at least two contexts where meta-definitions (structures that generate definitions) are encountered. The first is found in a proposal of enhancement of the DoNaLD notation given in [Beynon89]. Listing 7.3 is a proposed DoNaLD specification for displaying rungs of a ladder.

```
1. openshape ladder (int n, point a, b)
2. within ladder {
3.     line L = [n.a, n.a + (b-1)]
4.     if n > 1 then {
5.         shape Ladder = ladder(n-1, a, b)
6.     }
7. }

8. shape Ladder = ladder(7, {1,2}, {2,1})
```

Listing 7.3: A Proposed DoNaLD Specification of a Ladder

In the example, lines 1 to 7 defines a general definition of a ladder, ladder, and line 8 specifies an instance of ladder, Ladder.

The shape Ladder contains no self-referencing because the references created in the expression ladder(7, {1,2}, {2,1}) are all distinct from Ladder. The specification of ladder creates the variable Ladder/L which refers to the top rung, the variable

Ladder/Ladder to the remaining set of rungs, of which Ladder/Ladder/L is the topmost, etc. Note that the definition of Ladder in line 8 defines not only the variable Ladder itself but also a set of variables L, Ladder/L, ... , Ladder, Ladder/Ladder and so on.

The second context where meta-definitions are encountered is in the for-loop and the with-loop in the ARCA notation. Taking the with-loop as an example, the following loop:

```
with int 3 : I = 2, 3 do
     D!(2*I) = rot(D!2, I-1, v)
     D!(2*I-1) = rot(D!1, I-1, v)
od
```

is an abbreviation for:

```
D!4 = rot(D!2, 2-1, v)
D!3 = rot(D!1, 2-1, v)
D!6 = rot(D!2, 3-1, v)
D!5 = rot(D!1, 3-1, v)
```

The meta-definitions in both the DoNaLD and ARCA examples above may be interpreted as shorthands for fixed sets of definitions. That is to say, we may substitute sets of definitions for the meta-definitions. This substitution is possible because the parameters of ladder in the DoNaLD script and the values of the index I in the ARCA script are constant values.

The use of meta-definitions with abstractly defined parameters, as in the DoNaLD specification

```
shape Ladder = ladder(m, {x, y}, {2, 1})
```

raises more difficult issue of interpretation. Suppose that a meta-definition M with a parameter P produces f(P) ordinary definitions, then, when P changes, the total number of definitions will change. In a conventional definitive script, redefining P would cause a transition to a destination state differing from the original one only by the definition of P; at most one definition has been changed. Therefore, meta-definitions are not conventional definitions.

A meta-definition which involves an abstract parameter P can be viewed as an agent that effects a state transition when it observes a change in the definition of P. This means that a meta-definition is as much concerned with controlling transitions as with describing states.

7.2.2. Semi-evaluation

There is a semi-evaluation operator | ... | in the ARCA notation. This operator substitutes the current value of the expression within the two vertical bars to form a persistent definition. The semi-evaluation operator is useful in freezing the value of a variable. For example, in the definition

$$\text{sterling} = \text{foreign} \times |\ \text{rate}\ | - \text{charge}$$

the exchange rate can be fixed during a deal and the value of sterling depends on foreign and charge only. As indicated by this example, semi-evaluation can be a convenient way of assisting the construction of a state. It is particular useful within loopings where the current definition of the expression is difficult to obtain without consulting the transition history.

Definition involving the semi-evaluation operator can also be regarded as another form of meta-definition. This is because the definition stored is different from the definition typed in. A significant characteristic of this kind of definition is its context-sensitive nature – the meaning of the definition depends upon the current state. So semi-evaluation also acts like a simple agent to evaluate an expression in the current context, to modify the defining formula, and to effect a transition from the current state.

7.2.3. Inheritance

Since a definitive script does not necessarily specify a complete model, a definitive notation is very suitable for supporting inheritance. For example, we may want to extend DoNaLD so that it can define generic shapes (here, a generic shape does not

mean a class of shapes but a shape bearing certain properties that is generally not realisable due to the lack of some information). Listing 7.4 proposes one way in which we can specify how the properties of objects can be inherited from other objects. The based on syntax instructs the interpreter to copy all the definitions within the basic object into its own scope. So effectively, sqr is defined by the definitions in Listing 7.5.

```
openshape rectangle
within rectangle {
      real width, length
      real area = width * length
      point centre, NE, NW, SE, SW
      line N, E, S, W
      NE = centre + {width/2, length/2}
      NW = centre + {-width/2, length/2}
      SE = centre + {width/2, -length/2}
      SW = centre - {width/2, length/2}
      N = [NW, NE]
      E = [NE, SE]
      S = [SW, SE]
      W = [NW, SW]
}

openshape square based on rectangle
within square {
      real size
      width, length = size, size
}

openshape unitsquare based on square
within unitsquare {
      size = 1.0
}

openshape sqr based on unitsquare
within sqr {
      centre = {500, 500}
}
```

Listing 7.4: A Proposed DoNaLD Specification of Rectangular Objects

```
openshape sqr
within sqr {
      real width, length
      real area = width * length
      point centre, NE, NW, SE, SW
      line N, E, S, W
      NE = centre + {width/2, length/2}
      NW = centre + {-width/2, length/2}
      SE = centre + {width/2, -length/2}
      SW = centre - {width/2, length/2}
      N = [NW, NE]
      E = [NE, SE]
      S = [SW, SE]
      W = [NW, SW]
      real size
      width, length = size, size
      size = 1.0
      centre = {500, 500}
}
```

Listing 7.5: Specification of a Square

This scheme provides a dynamic inheritance method similar to the delegate technique in object-oriented programming [Lieberman86] – the properties of an object are inherited directly from another object instance rather than the properties of a class of objects are derived from another class of objects (cf.: "children inherit properties from their father" vs "babies inherit properties from adults").

At this stage, this inheritance scheme is only a tentative proposal for an extension to the DoNaLD notation. As object-oriented programming is becoming popular, inheritance is surely one of the issues that researches in any programming paradigm cannot overlook. Generic shapes such as the rectangular objects specified in Listing 7.4 are quite different in nature from traditional DoNaLD shapes. They are not generally realisable; they have to be interpreted with respect to observations of a different nature, viz observations of rectangular objects in general rather than any particular rectangular object.

The inheritance scheme we have suggested is a plausible attempt to relate definitive programming and object-oriented programming. This involves some extension of single-agent definitive programming. In particular, definition inheritance

can be regarded as another form of meta-definition. When the base template is changed, its dependent shapes must also be changed. Every generic shape specification can be interpreted as specifying an agent who monitors the base shape and redefines the derived shapes accordingly.

7.3. Input Management

A single-agent definitive system provides an interactive environment for program development. However, there is a distinction between an interactive environment for program development and an environment for developing general interactive programs. The notations and techniques discussed so far only enable us to interact through textual input of definitions; they do not enable us to develop interactive programs that make use of general mechanisms for user-input. To this end, we need to model input devices. Without means to model keyboard, mouse and time, we can hardly write programs with a reasonable user interface.

This section describes an extension to the Scout system so that mouse events and system generating events (such as a clock) can be interpreted. Two worked examples will be referred to in the course of discussion – a room viewer and a vehicle cruise control system. The DoNaLD specification of the room example was written by Edward Yung; the LSD specification of the vehicle cruise control system and its EDEN implementation were originally written by Ian Bridge. The graphical display and the mouse control were written on top of these two implementations using the extended Scout system. The interested reader may refer to Appendix F and G for complete detail.

7.3.1. The Room Example

A sample output of the room example is shown in Figure 7.1. The user may interact through the graphical interface in the following ways:

1. The menu options are self-explanatory.

2. The zoom window (the right-hand-side one) is partitioned into four regions – the four regions divided by the two diagonals; these four regions resembles the four menu options of the zoom position menus.

3. In the normal view (the window on the left), the table can be moved by direct manipulation. If the user presses a mouse button within the table area, drags the mouse and then releases it, the table will move by the same displacement of the mouse position.

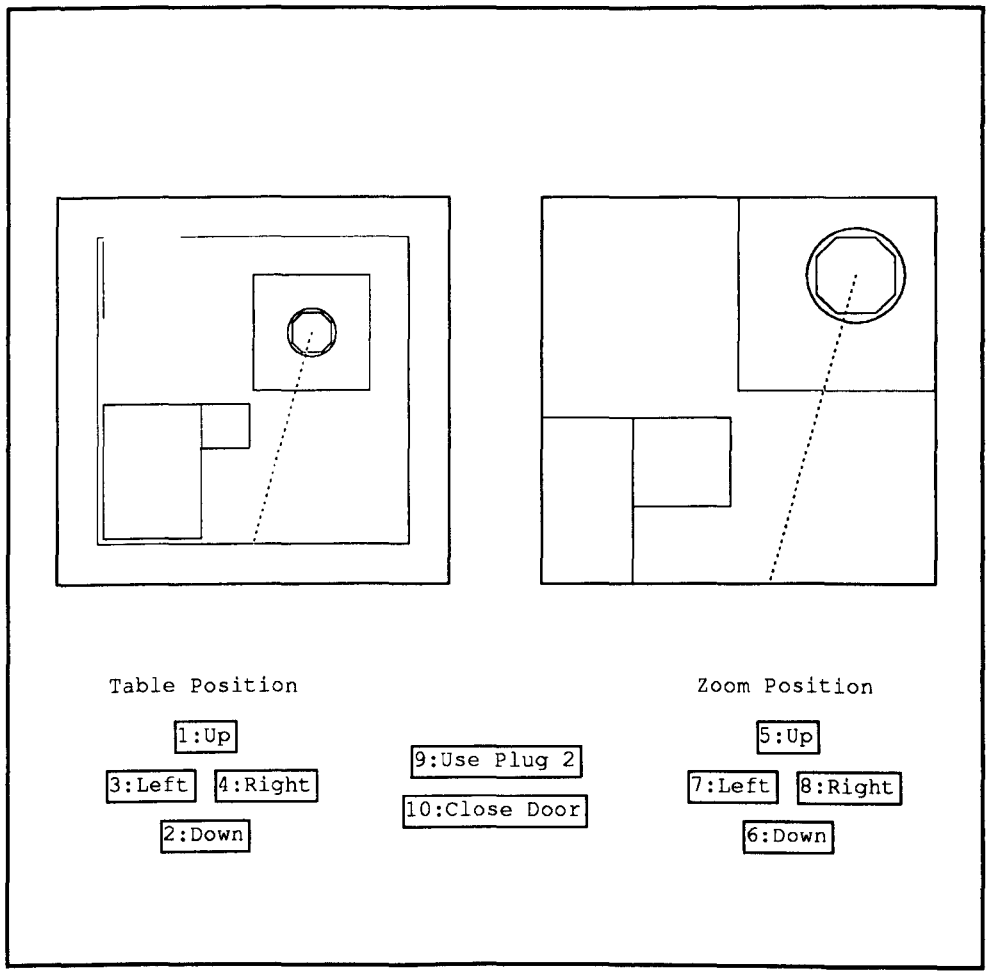


Figure 7.1: A Sample Output of the Room Viewer Example

7.3.2. The Vehicle Cruise Control Example

The vehicle cruise control example has been discussed in [BBY92]. The focus of [BBY92] is on an agent-oriented programming paradigm; the user-interface is discussed in the language of the LSD agent specification language (see next chapter). In this chapter, the EDEN implementations of some of the user-interfacing agents will be used to illustrate some techniques of input management in the Scout system.

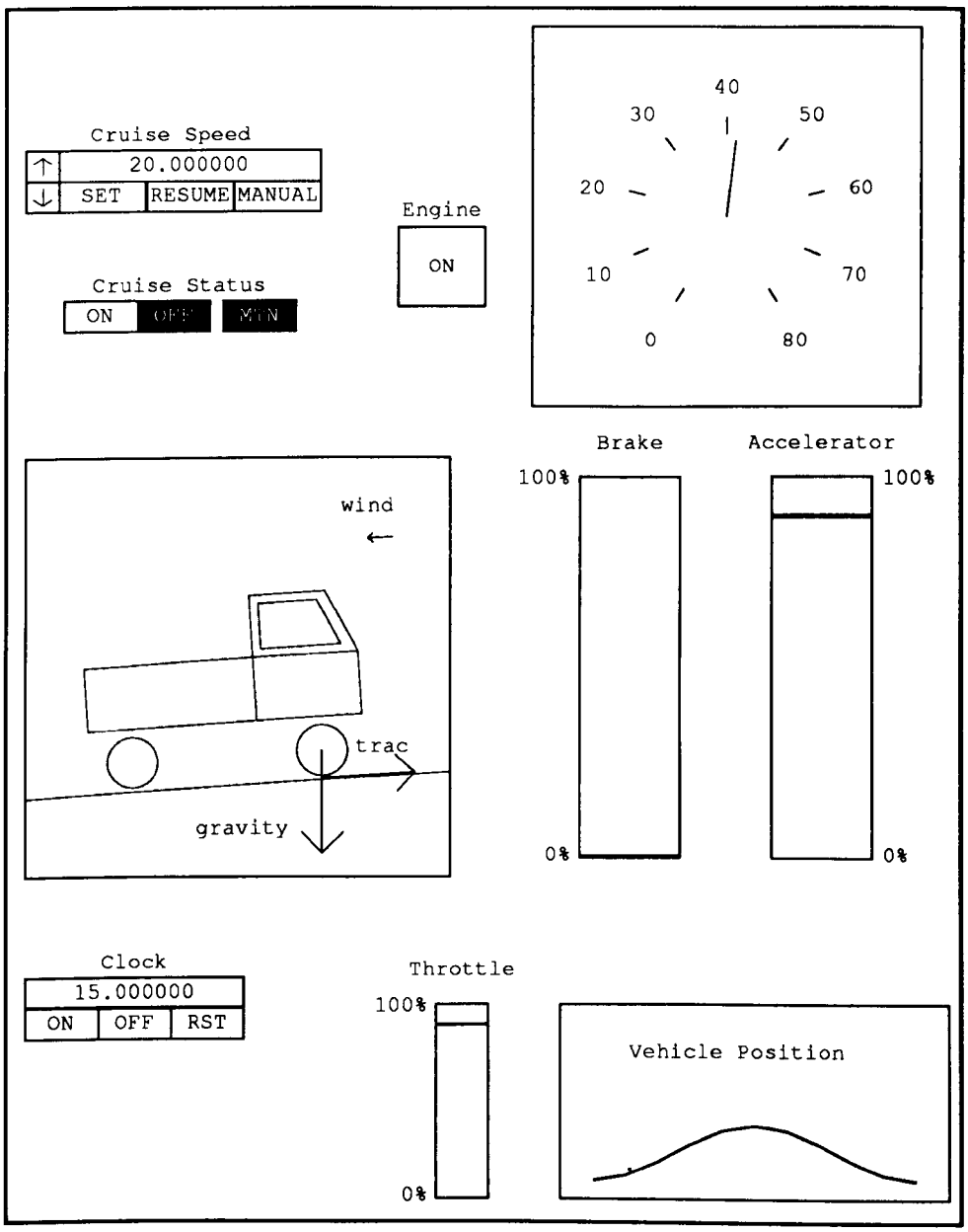


Figure 7.2: A Sample Output of the Vehicle Cruise Control Simulation

A sample output of the room example is shown in Figure 7.2. The user may interact through the graphical interface in the following ways:

1. Switch on or off the engine by pressing the ignition button. The ignition button is an example of toggle switch (see §7.3.4.2.).

2. Switch on or off the cruise controller by pressing the "ON" or "OFF" button on the cruise controller panel. The cruise controller switch is an example of radio buttons (see §7.3.4.4.).

3. Set and resume the cruise speed by pressing the "SET" and "RESUME" buttons respectively; switch to manual speed control by pressing the "MANUAL" button. The set of buttons – "SET", "RESUME" and "MANUAL" – also illustrates radio buttons.

4. Increase or decrease the cruise speed by pressing the buttons with the up arrow and the down arrow respectively. These two buttons are examples of duration-sensitive buttons (see §7.3.4.5.).

5. Change the position of the accelerator by pressing the mouse in the accelerator window. The nearer to the "100%" mark, the further the accelerator is depressed. The accelerator window is a variant of a menu button (see §7.3.4.3.).

6. Start, stop or reset the simulation clock by pressing the "ON", "OFF" and "RST" button in the clock panel respectively. They are implemented in the example as menu buttons (see §7.3.4.3.).

7.3.3. Extension to the Scout System

7.3.3.1. Considerations

The extension to the Scout system regarding input management takes into account the following points:

1) Limitations of the Original EDEN System

In the originally EDEN interpreter, the only user interaction is via typing in EDEN statements (including definitions). Upon received a definition or an action call initiated by the user, the interpreter will store the definition in its definition store and execute the action. All the actions triggered by the definition or invoked by the original action will then proceed. Not until all the triggered actions have been terminated will the system accept another statement. Under this scheme, there is no chance of processing any external input in the midst of a non-terminating loop. A problematic case is when a system clock has to be simulated. In a clocked system, such as the cruise control animation, there is no easy way to change the parameters, such as the accelerator position, while the clock is running[4].

2) Modes of Input

The aim of input is to initiate state transitions. In definitive programming, transitions are modelled by redefinitions. For this reason, we devise mechanisms to treat all modes of input as ordinary definitions. That is to say, a mouse pressing action, for example, will cause a variable to be (re)defined. Three modes of input are identified: user-generated events, e.g. mouse-pressed; system-generated events, e.g. clock updating; 'normal' channel of input, i.e. type-in EDEN statements.

3) Modes of Response

Ideally, we would like the system to be capable of different modes of response. For instance, an input may demand an immediate response (as in an interrupt, when the system suspends activities to service a user request) or may cause a change to the

---

[4] In the original EDEN cruise control simulation, as developed by Ian Bridge, the clock stops every 10 seconds to allow any possible change of definitions before the simulation continues (manually).

system without an immediate effect (as in polling activity, where the system monitors the values of input variables intermittently).

Redefinition of variables is a method of communicating state changes to the system that can be used for both interrupt and polling activities. Interrupt and polling activities correspond to different ways of associating actions with the redefinition of input variables. Since a definition may cause indivisible value changes and invoke EDEN actions in the same conceptual transition of state, implementing an input event as a definition can simulate the effect of an interrupt. In polling, actions are performed with reference to the current values of the input variables as and when appropriate.

4) Interrelationship between input management and Scout

It has been argued that separating input devices from application programs is inappropriate for modern user interfaces [Meads87]. It has also been argued that the Smalltalk "Model-View-Controller" (MVC) paradigm of an application may not take full advantage of the close relationship between output and input handler [Myers90]. In MVC, a program is separated into three parts: the *model* which embodies the application semantics, the *view* which handles the output graphics that show the model, and the *controller* which handles input and interaction. Unfortunately, programmers have found that the code for the controller and view are often tightly interlinked; creating a new view usually requires creating a corresponding new controller. In fact, both are often entwined with the model, so all three need to be rewritten.

In definitive programming, we also admit the close relationship between model, view and controller. A model is constructed by a definitive script; the realisation of the definitive script forms a display. The relationship between the model and the display is analogous to a mechanical linkage: a change to the model causes a simultaneous change to the display. Input handling also has a close relationship with both the display and the model. The meaning of pressing a button depends upon where the mouse is located; the input changes the model and in turn changes the display. In the Jugs

example mentioned in Chapter 5, the region inscribed in a menu window denotes the existence of a menu option. The button pressing action within that area should cause a change of the water levels in the jugs model and hence affect the jugs display. The information described in the Scout notation is useful for input management.

7.3.3.2. The Extension Plan

Based on the considerations above, the extension plan to the Scout system involves the following:

1. Since the meaning of an activity of an input device depends on the location of the pointer device, Scout window has a new attribute *sensitivity*. In the current design, this field is only a boolean value indicating whether input is accepted inside the window. Ideally, *sensitivity* may also be used to specify which kind of input is acceptable.

2. In view of 1), the EDEN/X Window interface EX has to be able to generate a definition upon an input action within a sensitive Scout window. In order to assist the interpretation of the input, the variable name to be defined must be related to the Scout window name. In our current implementation, the kinds of input EX manages are pressing and releasing of mouse button and key-press on keyboard. Mouse movement leads to such frequent generation of definitions that system performance becomes unacceptably slow, and mouse movement is not currently managed.

   The variable name to be defined is determined by which region the mouse is in. Consider a button pressing or releasing action. If the mouse is within a DoNaLD or ARCA window, the variable name would be the Scout window name concatenates with "_mouse"; if the mouse is within a text window, it would be the Scout

window name concatenated with "_mouse_" followed by the box number[5]. When the mouse action occurs, the value assigned to the appropriate variable records the nature and the location of the mouse action. Currently, the value is a 5-tuple of (*button*, *type*, *state*, *x*, *y*),

where *button* = the button number pressed or released;

type = the button action (4 = pressed, 5 = released);

state = the state before the button action occurred (shift (+1), caplock (+2), control (+4), meta (+8) and was pressed (+256)). For example, if a button is released while the shift and control keys are depressing, *state* will be 1 + 4 + 256 = 261;

x, *y* = the x- and y- coordinates of the mouse in the coordinate system of the window in which the mouse action occurred.

As with mouse events, a stroke on the keyboard will generate a definition. Instead of "_mouse" or "_mouse_" followed by a box number, the variable name will end with "_key" or "_key_" followed by a box number. The value defined will also be a 5-tuple: (*key*, *type*, *state*, *x*, *y*), where *key* is the ascii code of the key pressed.

As an example (taken from the vehicle cruise control simulation in Figure 7.2), consider the Scout window is defined as follows:

---

[5] A text window may consist more than one boxes whereas exactly one box constitutes a graphic window.

```
point    brakeOrg = {225, 250};
integer  BAwidth = 50;
integer  BAlength = 200;
window   brakePedal = {
         type:      DONALD,
         box:       [brakeOrg, brakeOrg + {BAwidth, BAlength}],
         pict:      "BRAKE",
         xmax:      100,
         ymax:      100,
         border:    1,
         sensitive: ON
};
```

A mouse click (press and release) in this window will typically generate the following definitions:

```
brakePedal_mouse = [1, 4, 0, 50, 70];
brakePedal_mouse = [1, 5, 256, 50, 70];
```

3. The EDEN interpreter has to be able to manage definitions coming from different sources. The definitions generated by EX are sent to EDEN via a *message queue*, a UNIX system V inter-process communication method, whereas the type-in definitions come in through a pipeline. An EDEN action may also generate definitions (system-generated events) and send these to the EDEN interpreter via the message queue. This definition will then be processed in the next time slot[6] of the interpreter. The input management for the EDEN interpreter has to be modified so that input accepted from the pipeline and the message queue is interleaved.

[6] In a *time slot*, the EDEN interpreter will process an EDEN statement and all its consequent actions.

7.3.4. Input Handling Techniques

The following sub-sections describe how the definitions generated by input events can be combined with different patterns of EDEN actions to animate a timer and to implement different kinds of buttons.

7.3.4.1. Push Button

A push button is one which is logically *true* when it is pressed and is *false* when it is released. This is the simplest form of button. It can be animated by a definition as simple as:

```
ButtonStatus is PB_mouse[2] == 4;
```

This defines ButtonStatus to be true when a mouse button is pressed in the Scout PB window, and false otherwise.

7.3.4.2. Toggle Switch

A toggle switch is one which has an initial state. Every time a button is pressed, the state of the toggle switch reverses; releasing a button has no effect on the toggle switch's state. The previous state of a toggle switch has to be remembered and an initial state has to be defined. Typically, a toggle switch is animated as follows:

```
engineStts = esOn;
proc chgEngineStts : ignition_mouse_1 {
        if (ignition_mouse_1[2] == 4) {
                engineStts = (engineStts == esOn) ? esOff : esOn;
        }
}
```

In this example, the engine status (engineStts) is initially esOn. The engine status will alternatively change to esOff and esOn whenever a button is pressed in the ignition window.

7.3.4.3. Menu Buttons

A menu button is a button that invokes an action when it is pressed (or released depending on the design). It is like a door bell which will start a melody when the button is pressed; the melody will continue even if the button is released (releasing the button has no effect on the door bell). This kind of button is often used in menu selection. For example in the room viewer example (in Figure 7.1), pressing the "zoom up" menu option (the zoomUp window) once will move the zooming area up by 100 units. The implementation of the "zoom up" option is as follow:

```
proc zoomUp_action : zoomUp_mouse_1 {
        if (zoomUp_mouse_1[2] == 4) {
                zoomPos = pt_add7(zoomPos, [100, 0]);
        }
}
```

7.3.4.4. Radio Buttons

Radio buttons are defined by a set of buttons amongst which exactly one of them is depressed at any time. The pressing of one button will cause another button which is currently selected to be released. The following shows an example of a set of three radio buttons (RB1, RB2 and RB3) with the initial condition of button RB1 is on. This scheme requires the knowledge of the current values of the buttons but minimises the updating of variables[8].

---

[7] pt_add() performs a vector sum of the two argument points.

[8] Alternatively we can define
proc update_buttons1 : rb1_mouse { **if** (rb1_mouse[2] == 4) { RB1 = 1; RB2 = 0; RB3 = 0; } }
and so on. The method shown in the main text is preferred because it minimises the number of variables to be redefined.

```
RB1 = 1; RB2 = 0; RB3 = 0;

proc set_RB1 : rb1_mouse { if (rb1_mouse[2] == 4 && !RB1) RB1 = 1; }
proc unset_RB1 : RB2, RB3 { if ((RB2 || RB3) && RB1) { RB1 = 0; } }

proc set_RB2 : rb2_mouse { if (rb2_mouse[2] == 4 && !RB2) RB2 = 1; }
proc unset_RB2 : RB1, RB3 { if ((RB1 || RB3) && RB2) RB2 = 0; } }

proc set_RB3 : rb3_mouse { if (rb3_mouse[2] == 4 && !RB3) RB3 = 1; }
proc unset_RB3 : RB1, RB2 { if ((RB1 || RB2) && RB3) RB3 = 0; }
```

7.3.4.5. Duration-Sensitive Button

A duration-sensitive button is essentially a push button. The reason for putting the duration-sensitive button in a separate category is that its duration of pressing, rather than its logical state, is significant. The implementation of a duration-sensitive button is therefore different from that of a push button.

```
incrBtn = pbUp;
proc incCrSpeed : incrBtn, crUpWin_mouse {
        if (crUpWin_mouse[2] == 4) {    /* button pressed */
                SendToEden("incrBtn = pbDown;\n");
                if ((cruiseStts != csOff)  && (cruiseSpeed_mph < maxCruiseSpeed_mph))
                        cruiseSpeed_mph = cruiseSpeed_mph + 1;
        } else { /* button released */
                if (incrBtn == pbDown) incrBtn = pbUp;
        }
}
```

In the above example (cf Figure 7.2), when a mouse button is pressed in the crUpWin window ("increase cruise speed" button), the setting for the cruise speed will be increased by 1 mph repeatedly so long as the button remains in a down position. SendToEden() is an EDEN function which will send the argument string to the EDEN interpreter itself via the message queue. Since the definition denoted by the argument string redefines incrBtn, a triggering variable of the incCrSpeed action itself, the variable cruiseSpeed_mph will keep on increasing in every time slot until the button is released.

7.3.4.6. Clocking

The technique used to implement the duration sensitive buttons is also applicable to the simulation of a system clock. The following shows how a chime may be implemented. This chime will print a line "Bell!" every 5 seconds.

```
chime = 0;
proc clock_watcher : clock {
        if (clock – clock_init) >= 5) {
                clock_init = clock;
                chime++;
        }
        SendToEden("clock = "//str(time())//";\n");[9] /* update clock */
}
proc bell : chime { if (chime) writeln("Bell!"); }
clock = clock_init = time(); /* set clock to current time; start the clock */
```

The above techniques are sufficient for most, but not all, kinds of interaction in the room viewer and cruise control examples. These techniques only use the button pressing or releasing status. Other information, such as the position of the mouse in the window, is not used. There are cases in the room viewer and cruise control examples where the positional information is used. For example, movement of the table in the room (see Figure 7.1) is related to the displacement between the mouse button pressing and button releasing positions; the position of the mouse when pressing a button in the accelerator window (see Figure 7.2) determines how far the accelerator is depressed.

There are many techniques for interaction that are currently in use or are desirable. The two worked examples illustrate basic principles upon which a large

---

[9] SendToEden() makes use of message queue (one of the System V IPC methods) to send a definition to EDEN itself rather than directly executing the definition. This is because direct execution will block the execution of other definitions and actions whereas messages will be processed later when other actions are done.

number of techniques can be built up, and demonstrate that definitive programming can easily produce elegant user-interfaces.

7.4. Summary and Conclusion

There are two possible directions in which to generalise single-agent definitive systems. One is to develop more complex underlying algebras; the other is to introduce agents. This chapter starts by exploring the first possibility. We have investigated the Admira prototype. An Admira script can be interpreted as part pure definitive script and part functional specification of the algebra underneath the definitive script. Admira shows that, by having an appropriate underlying algebra, a definitive script can describe a complex state which encapsulates many states and transitions. However, when we increase the computational power of the operators in the underlying algebra in this way, we sacrifice clarity in state-based interpretation of the associated scripts. For example, Miranda has complex operators, but a Miranda script has an obscure state-based interpretation. In contrast, DoNaLD has simple operators but a DoNaLD script has a clear state-based interpretation.

In the Definitive State-Transition model, the state description should be determined by the mode of observation. This allows us to gain maximum understanding of the real-world system. Definitive programming focuses on the description of the relationship between observable properties of the real world; developing powerful underlying algebra is of secondary importance. This means that developing powerful underlying algebra is no substitute for the introduction of agents.

Some existing and proposed features of definitive notations like loops, semi-evaluation and inheritance are agent-like. In particular, we can view EDEN actions used for input management as programmable agents. Agents for governing the state transitions are important, but have to be used in a disciplined way. Improper use of EDEN actions, for example, can make the difference between principled definitive

programming and an anarchic form of procedural programming. In the next chapter, we shall discuss an agent-oriented definitive system formally.

8

Agent-Oriented Definitive Programming

The central concept of definitive programming is modelling a state by a set of definitions. The expressive power of definitions is enhanced by the techniques we have built up for integrating several definitive notations. But whatever descriptive power a set of definitions has, it is meant to describe only one state. For a typical programming task, it is insufficient to have states without transitions.

EDEN procedures and actions can program the transition between states, but EDEN's control structure is fundamentally imperative. This thesis claims that definitive programming is an exploratory programming paradigm. We would like to see that a definitive program is adaptable to the RUDE cycle of software development. A definitive program should be efficient to *Run*, easy to *Understand* and *Debug*, and on-line *Editable*. We have shown in the previous chapters that definitive states fulfil these requirements, and we would like to see the transition control structure over definitive

states exhibits similar advantages. Although EDEN is the best developed software tool for definitive programming so far, its imperative control structure makes it a poor choice for specifying transitions.

This chapter introduces an agent-oriented approach to transition control of definitive states. The LSD agent-protocol specification language was first proposed by Beynon and Norris in [Beynon86b, BN87] and had its major development by Slade [Slade89]. The purpose of the LSD language is to specify the privileges of the agents to act upon one another in a system. An LSD specification describes an essential part, but not all, of the behaviour of the agents. For this reason, an LSD specification is not an executable program, and it is not expected to be executable. This chapter reviews the LSD programming environment, describes its development since 1987 and gives suggestions for its future development.

LSD was developed for concurrent programming; it was also designed to model agent activities. Therefore, a program derived from an LSD specification should be efficient to run; an LSD specification should be easy to understand and debug. However, the prototyping facility for animating an LSD specification is not as convenient as it might be. The animation of an LSD specification relies on a programming tool, namely the ADM definitive language [BSY88]. The current transformation process from an LSD specification to an ADM program has a few limitations:

1. An LSD specification has an ambiguous interpretation but an ADM program has unambiguous operational semantics. The transformation from LSD to ADM requires additional information about what we have called *scenario information* or *simulation decisions*. This information is not part of an LSD specification. For this reason, the first problem of the transformation is that it cannot be automated.

2. Since the ADM language has only the integer data type, the LSD specifications that can be transformed into an ADM program are just those that require integer, or

more generally enumerated data types. This limits the range of application LSD can specify.

3. The only ADM output channel is a print-statement. ADM is not an ideal environment for the visualisation of the current definitive state.

Our research target in connection with LSD is therefore to seek ways of improving the usability of LSD. To this end, my contribution in this thesis has been:

1. to evaluate and suggest improvements for the LSD notation. These suggestions should make LSD more expressive, and at the same time, preserve the essential characteristics of LSD.

2. to write an ADM-to-EDEN translator. By using automatic translation of an ADM program to an EDEN program in conjunction with other definitive scripts for graphical presentation, a graphical animation of the ADM program can be obtained using the Scout system.

Figure 8.1 shows schematically how we may currently animate an LSD specification. The sample outputs in Figure 8.1 and 8.2 are extracted from a worked railway station simulation example.

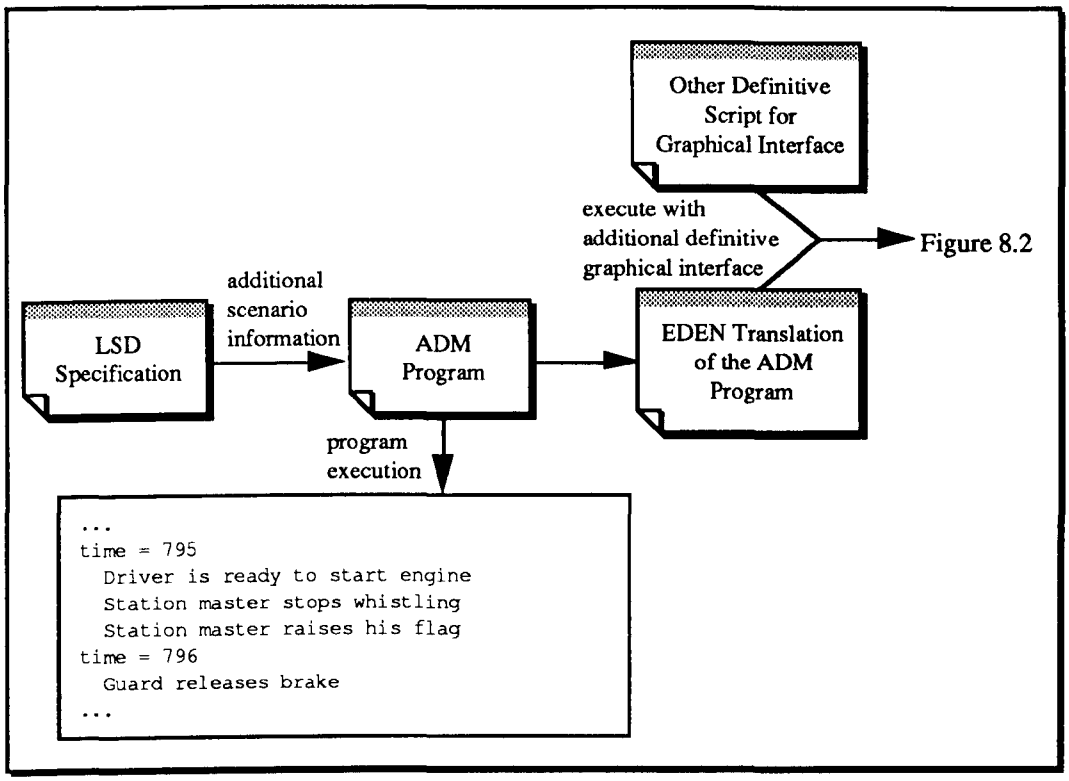


Figure 8.1: Procedures for Animating an LSD Specification

Figure 8.1 should be interpreted as follows. An LSD specification is first transformed into an ADM program. The transformation process involves decision making for determining the exact behaviour of the agents from their privileges specified in LSD. The transformed ADM program can be executed directly by an ADM interpreter. A typical output from the ADM program is a textual commentary recording the events that occur during the animation. Alternatively, the ADM program can be translated automatically into an EDEN program using a translator I have written for the purpose. The translated EDEN program can then be supplemented with other definitive scripts, such as SCOUT and DoNaLD. These additional definitions may serve as a graphical interface to the simulation. Graphical interpretation of the current definitive state, such as Figure 8.2, may, therefore, be obtained in addition to a commentary.

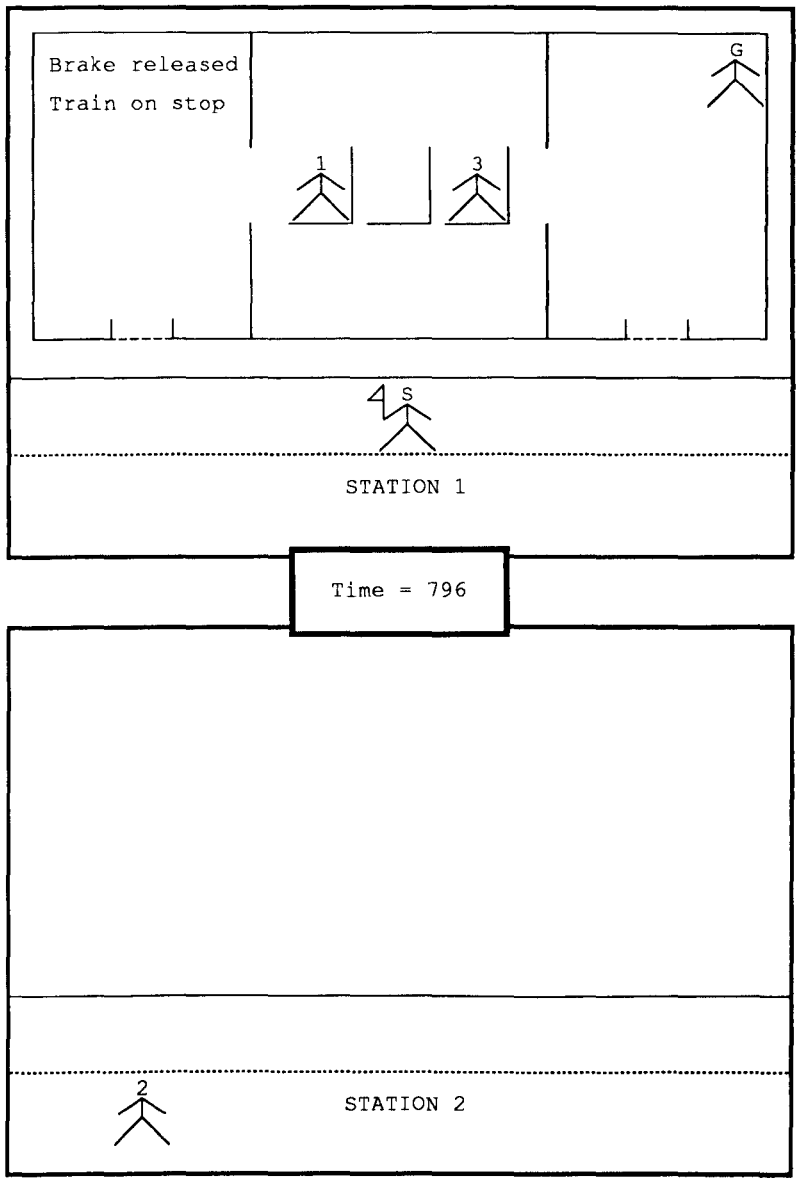


Figure 8.2: Sample Display of the Railway Station Simulation

The suggestions for enhancing the LSD notation given below, combined with the ADM-to-EDEN translator prototype will greatly reduce the time needed to produce an executable program from an LSD specification. In this way, the LSD programming environment will be better adapted for exploratory programming.

8.1. The Railway Station Simulation

A railway station simulation will be used to illustrate the process of animating an LSD specification. In this simulation we animate a train with some travellers commuting between two railway stations. The train arrival/departure protocol takes the following form:

As the train approaches the station

the guard applies the brake to stop the engine

After the train has waited at the station for an appropriate interval of time

the station-master checks and shuts the doors

Meanwhile passengers are alighting and boarding the train

When all doors are closed

the station-master whistles to call the attention of driver and guard

The guard waits for the station-master to raise his flag

to signal the release of the brake

The driver gives the ready signal to the station-master who raises his flag

After the brake is released

the guard signals to the station-master by raising a flag

The station-master signals the driver to start the engine.

The agents involved are identified from the above protocol, and each agent is described by an LSD agent. Some agents involved in the simulation are personnel who are continuously determining their next action; they are the *station-master*, the *guard*, the *driver* and the *passengers*. Some agents are passive objects that are manipulated by other agents or are routinely doing a job; they are the *train* and the *clock*. A door should also be a passive object, but at most one person can pass through a door at a time, and in order to simplify the protocol for door use by passengers, a *door* agent in our specification includes a mechanism to choose which passenger should pass through when several attempt to pass through simultaneously.

Having identified the agents, we identify what attributes they possess (the *state* variables), what they can perceive in the environment (the *oracle* variables), what can they change in the environment (the *handle* variables), what knowledge they can derive from the things they perceived (the *derivates*), and what privileges make up their protocols. As an example, Listing 8.1 shows the LSD specification of the station-master agent. The complete LSD specification of the railway station simulation can be found in Appendix H.

```
agent sm() {                                    // The station master:
state       (time) tarrive = !Time!;            // registers time of arrival
            (bool) can_move = false;            // determines whether the driver can start the engine
            (bool) whistle = false;             // controls the whistle
            (bool) whistled = false;            // remembers whether he has blown the whistle
            (bool) sm_flag = false;             // controls the flag
            (bool) sm_raised_flag = false;      // remembers whether he has raised the flag
oracle      (time) Limit                        // knows the time to elapse before departure due
            (time) Time;                        // knows the current time
            (bool) guard_raised_flag;           // knows whether the guard has raised his flag
            (bool) driver_ready;                // knows the driver is ready
            (bool) around[d]; (d = 1 .. number_of_doors)
                                                // knows whether there's anybody around doorway
            (bool) door_open[d]; (d = 1 .. number_of_doors)      // the doors status
handle      (bool) can_move, whistle, whistled, sm_flag, sm_raised_flag;
            (bool) door_open[d]; (d = 1 .. number_of_doors)      // partially controls the doors
derivate    (bool) ready = ∧ (¬door_open[d]) | d = 1 .. number_of_doors);
                                                // monitors whether all doors are shut
            (bool) timeout = (Time – tarrive) > Limit;      // monitors whether departure is due
protocol    door_open[d] ^ ¬around[d] -> door_open[d] = false (d = 1 .. number_of_doors)
            ready ^ timeout ^ ¬whistled -> whistle = true; whistled = true[1]; guard(); whistle = false
            ready ^ whistled ^ ¬sm_raised_flag -> sm_flag = true; sm_raised_flag = true
            sm_flag ^ guard_raised_flag -> sm_flag = false
            ready ^ guard_raised_flag ^ driver_ready ^ engaged ^ ¬can_move -> can_move = true
}
```

Listing 8.1: An LSD Specification of a Station-Master

8.2. Terminology in the LSD Notation

LSD is a specification language for concurrent applications. Although the terminology used in LSD has been changed since the first discussions of it in the papers

[1] See §8.3 for the reason of underlining these definitions.

[Beynon86b, BN87], the principle underlying LSD remains unchanged. LSD describes a system by describing the behaviour of the individual agents involved in the system.

In Mike Slade's definition of LSD [Slade89], an agent description takes the following form:

```
agent agent_name (parameter_list) {
oracle list_of_oracle_variables
state list_of_state_variables
derivate list_of_derivate_variables
protocol list_of_guarded_actions
}
```

A *guarded_action* takes the form

```
boolean_condition -> list_of_actions
```

and an *action* is a *definition*, an *instantiation* or a *deletion* of an agent.

An LSD specification for an agent describes:

- the aspects of the system state to which it can respond – its *oracle* variables;
- those aspects it can conditionally change – its *state* variables;
- the circumstances under which state-changing actions can be performed – its *protocol*;
- definitions which can express the different ways in which agent actions are to be interpreted in state-transition terms according to the context – its *derivate* variables.

Slade's version of LSD has already deviated from that of [Beynon86b] and [BN87] in that the term *agent* is used instead of *process* (a term originally derived from SDL [BN87]). Both agents and processes describe strands of activity in a computation. The reason for the change is that *process* suggests a circumscribed pattern of state changes undergone whilst *agent* suggests active changes whose effect is yet to be circumscribed.

From previous experience of communicating the LSD notation to various audiences, we have found that the original terminology of LSD is also confusing in other respects:

1. *Types of variables*

There were three kinds of variable qualifiers in the original LSD:

state variable – variable that can be (re)defined by the agent;

oracle variable – variable whose value is known by the agent;

owned variable – variable owned by the agent.

Owned variables were identified by a hatch sign (#) preceding the identifiers. More than one qualifier might be connected with the same variable name.

Changes to these qualifier conventions were introduced in [BBY92]. Since the qualifier "*state*" suggests variable "determining the state" of an agent, and hence owned by it, the qualifier conventions have been changed to the following:

state variable – variable owned by the agent (was owned variable);

oracle variable – variable whose value is known by the agent (as before);

handle variable – variable that can be (re)defined by the agent (was *state* variable).

As before, the same variable name can have more than one qualifier.

2. *The term protocol*

There is a sense in which the term *protocol* suggests a rigid pattern of execution for the guarded actions. In fact, the protocol of an LSD agent should be interpreted according to the following steps:

1. All the guards are evaluated.

2. If at least one of the guards is true then a guarded action is chosen arbitrarily, otherwise the guards are re-evaluated.

3. The action list associated with the chosen guard is executed sequentially.

4. The procedure is repeated.

This interpretation scheme has a non-deterministic nature. There is no indication of the likelihood with which a particular guarded action is chosen; there is also no indication of the delays between actions in the action list. So the LSD specification specifies what the agents *can* do rather than what the agents *will* do under certain circumstance. The actual behaviour may differ according to the simulation decisions made. The term *privilege* more accurately describes a guarded action.

The term *protocol* remains meaningful in the sense that the group of guarded actions does in fact describe the privileges in an agent's protocol. For example in a railway station simulation, the LSD specification is describing a train arrival/departure protocol. Therefore the term *protocol* is still acceptable though *privilege* is arguably a better word. For the present, we have decided not to change terminology in this respect. To prevent too many versions of LSD notations being in circulation, the term *protocol* is used in this thesis. As a result, the LSD examples in this thesis, such as Listing 8.1, use the same version of LSD as in [BBY92].

8.3. Transformation from LSD to ADM

An ADM program consists of a set of entity descriptions and instances of entities. An entity consists of a set of definitive variables and a set of actions. Examples of entity descriptions are:

```
entity clock()
{
definition
      time = 0,
action
      true -> time = (|time| + 1) % (3600 * 24)
                              // 3600 * 24 = number of seconds in a day
}

entity alarm()
{
definition
      switch = true,
      alarm_time = 8 * 3600, // 8 o'clock
      alarm = switch && (time - alarm_time) % (3600 * 24) >= 0 &&
            (time - alarm_time) % (3600 * 24) < 20
                                              // alarm on for 20 seconds
action
      alarm
         print("BEEP!")
            -> beep()
}
```

Listing 8.2: ADM Entity Descriptions of Alarm() and Clock()[2]

ADM has an unambiguous operational semantics. When an ADM program is started executing, the definitions in the definition sections of the entities are stored in the ADM's Definition Store and actions in the action sections are stored in the ADM's Action Store. Then all guards of the actions are evaluated in the context of the script associated with the definition store. If a guard is true, the message in the print statement will be displayed and the associated action(s) are stored in a Run Set. After all guards are evaluated:

- if the Run Set contains no actions, execution terminates;
- if the Run Set contains conflicting actions, for examples multiple redefinitions of the same variable, execution halts;
- otherwise, all actions are executed in parallel.

---

[2] Taken from [Slade89]

An action can be a redefinition, an entity instantiation, an entity deletion or a stop action. An entity instantiation will cause the definitions and actions of the new entity to be stored in the definition store and the action store respectively. Similarly, an entity deletion will remove the definitions and actions belonged to the entity from the definition store and the action store.

When all the actions in the Run Set have been executed, the ADM system is in a new state. The system has now completed an execution cycle. The process is then repeated.

Since the LSD notation does not specify the precise nature of interaction and synchronisation between agents, an LSD specification describes a family of possible behaviours. To execute an LSD specification, synchronisation details (we have called them *simulation decisions* or *scenario information*) must be added. Slade identified a number of issues for transforming an LSD specification to an ADM program [Slade89]. These issues concern the following questions:

(Q1) How accurate are the oracle variables? In real life, there are often some passengers who are travelling on the wrong train. This may be due to the passengers' false perception of time or platform number. In the simulation, the accuracy of the oracle variables reflects the degree of faithfulness in the model of the railway system.

(Q2) How frequently are the guards evaluated? For example: how frequently does the station-master check for time out?

(Q3) Are there any parallel actions?

(Q4) What are the response times and delays between actions?

(Q5) Are guards mutually exclusive? In other words, is an agent privileged to start more than one series of actions at the same time?

By applying the transformation techniques provided by Slade in addressing these issues, we have transformed the station-master agent in Listing 8.1 to an sm entity as in Listing 8.3.

```
entity sm() {
definition
        whistle = false,
        whistled = false,
        sm_flag = false,
        sm_raised_flag = false,
        can_move = false,
        ready = !door_open[1] && !door_open[2],
        tarrive,
        timeout = (Time - tarrive) > Limit,
        level = 0,
        init = true
action
        init
                -> tarrive = |Time|; init = false,
        door_open[1] && !around[1]
            print("Station master shuts door 1")
                -> door_open[1] = false,
        door_open[2] && !around[2]
            print("Station master shuts door 2")
                -> door_open[2] = false,
        ready && timeout && !whistled
            print("Station master whistles to call guard")
                -> whistle = true; whistled = true; guard(); level = 1,
        level == 1
            print("Station master stops whistling")
                -> whistle = false; level = 0,
        ready && whistled && !sm_raised_flag
            print("Station master raises his flag")
                -> sm_flag = true; sm_raised_flag = true,
        sm_flag && guard_raised_flag
            print("Station master lowers his flag")
                -> sm_flag = false,
        ready && guard_raised_flag && driver_ready && engaged && !can_move
            print("Train can move now")
                -> can_move = true
}
```

Listing 8.3: ADM Specification of the Station-Master Entity

This sm entity reflects the following assumptions about the behaviour of an ideal sm agent: the agent has immediate and correct knowledge of its environment (oracle variables), quickest response time[3] and minimal delay between actions in the action

[3] *Quickest response time* and *minimal delay* are with respect to the limit of ADM. That is an action will take place in the next ADM time slot when the guard becomes true; sequential actions will be performed in consecutive slots.

lists. Although these assumptions are not entirely realistic, they lead to the simplest implementation of the sm entity in programming terms.

8.4. An Evaluation of the LSD Notation

Deutsch suggests a scenario-oriented approach for programming [Deutsch89]. A scenario typically describes a stimulus-response relationship, a behaviour pattern that would be visible to a system user. Deutsch argues that this kind of description will enhance communication between non-computer experts, such as users and customers, and the software engineers who are developing the system. An LSD specification resembles a scenario-oriented specification in that a guarded action is similar to a scenario in Deutsch's sense – the guards are the stimuli and the associated actions are the responses. By Deutsch's criteria, LSD is a good specification language. In comparison, an ADM program is less concise and comprehensible than the corresponding LSD specification. For example in the Railway Station simulation example, the ADM program is about twice the length of the LSD specification and is far less readable. This is the cost of putting precise operational details into a specification.

Programming using LSD and ADM reveals a tension between intelligibility and ambiguity. Though neglecting operational details makes the operational interpretation of an LSD specification ambiguous, it also means that the specification can intelligibly describe a family of simulations. An LSD specification has to be transformed into an executable form, but we still advocate that it is a better practice to specify the program in LSD first. The operational ambiguity is relatively easily resolved by systematically addressing the simulation issues, whilst intelligibility is harder to achieve.

During the transformation to ADM, the information essential for the determination of operational behaviour is inserted. The tension between intelligibility and ambiguity means that the transformation process cannot be done automatically. It seems promising though that a hidden text annotation of an LSD specification can be

transformed automatically into an ADM program without destroying the intelligibility of the LSD notation.

The rest of this chapter evaluates the possibility of automatic translation. The following sub-sections show some suggestions for enhancing and annotating the LSD notation. When these suggestions are implemented, it is entirely possible that an annotated LSD specification can be transformed automatically into an ADM program.

8.4.1. Grouping Guarded Commands

In the present design of LSD, the guarded actions are grouped by agent in such a way that only one guarded action is chosen arbitrarily if more than one guard in the protocol is true. In general, an agent may be capable of performing two uncoordinated actions simultaneously, for example, as when a person is walking and clapping at the same time. This argues for the introduction of a hierarchical grouping of actions corresponding to a decomposition of an agent into sub-agents. In this way, more than one guarded action (at most one from each group representing a sub-agent) can be executed concurrently.

8.4.2. Parallel Action Specification

Notice that the underlined definitions in Listing 8.1 should, in principle, be executed in parallel. This is correctly reflected in the corresponding ADM entity. However, the standard interpretation of an LSD guarded action restricts the actions in the action list to be executed sequentially (cf [Slade89]). This means that the transformation of the station-master agent is not entirely faithful. On the other hand, the current LSD notation has no provision for specifying synchronised actions.

To deal with synchronised actions, a new parallel action separator could be added to LSD to specify parallel execution of actions in the action list. Associated with this change, brackets are needed to disambiguate the grouping of parallel and sequential

actions. With this parallel actions enforcing technique, we could then specify agents such as the following swapping agent:

```
agent swap() {
state     done = false
oracle    a, b, done
handle    a, b, done
protocol
          !done -> (a = |b| // b = |a|); done = true
}
```

Listing 8.4: A Swapping Agent Illustrating Parallel Actions

This cannot otherwise be specified so concisely.

8.4.2.1. Limitation of LSD for Specifying a Swapping Agent

Listing 8.5 and 8.6 are two attempts to swap the values of a and b; both attempts fail. Listing 8.5 fails because the action list associated with the chosen guard is executed sequentially: the result of execution will be that both a and b will be defined as the original value of b. Listing 8.6 fails because initially both guards are true, just one of the guarded action list is chosen arbitrarily rather than both, with the result that both a and b will acquire the original value of either a or b non-deterministically.

```
agent swap1() {
state     done = false
oracle    a, b, done
handle    a, b, done
protocol
          !done -> a = |b|; b = |a|; done = true
}
```

Listing 8.5: A Swapping Example (1st Attempt)

```
agent swap2() {
state       adone = false
            bdone = false
oracle      a, b, adone, bdone
handle      a, b, adone, bdone
protocol
            !adone -> a = |b|; adone = true
            !bdone -> b = |a|; bdone = true
}
```

Listing 8.6: A Swapping Example (2nd Attempt)

```
agent swap3() {
state   done = false, temp
oracle  a, b, temp, done
handle  a, b, temp, done
protocol
        !done -> temp = |a|; a = |b|; b = |temp|; done = true
}
```

Listing 8.7: A Swapping Example (3rd Attempt)

Listing 8.7 is the third attempt to the problem. The values of the variables a and b are successfully swapped by using a temporary variable, as in a conventional procedural program. Cognitive interpretation of the actions of the first three attempts can be made by imagining the agents are trying to move items between boxes, an item at most can be put in each box at any time. Swap3 exchanges the items inside the boxes a and b by moving one item at a time, this method requires one extra box but needs one hand only; swap1 and swap2 try to exchange the items simultaneously, it needs two hands to pick up the two items and then replace them in position at the same time. Listing 8.8 shows how this two-hand idea may be implemented in LSD. This implementation is very inefficient because it involves i) many variables, ii) instantiation and deletion of agents and iii) relatively complex synchronization between agents atob and btoa. Furthermore, this implementation still cannot guarantee simultaneous execution of actions. The fundamental weakness of the LSD agent is the inability to perform two actions in parallel by an agent. This restricts what an agent could do; it is also an undesirable feature in terms of concurrency (we would like to do as many actions in parallel as possible).

```
agent swap() {
state    done = false, start = true
oracle   start, adone, bdone
handle   start, done
protocol
      start -> atob(); btoa(); start = false
      adone && bdone -> delete atob(); delete btoa(); done = true
}

agent atob() {
state    a_val = |a|, held_a = true, bdone = false
oracle   a, a_val, held_b
handle   held_a, b, bdone
protocol
         held_b -> b = |a_val|; bdone = true
}

agent btoa() {
state    b_val = |b|, held_b = true, adone = false
oracle   b, b_val, held_a
handle   held_b, a, adone
protocol
         held_a -> a = |b_val|; adone = true
}
```

Listing 8.8: A Swapping Example (4th Attempt)

8.4.3. Call-by-Reference Parameter

Another problem with all five attempts to specify a swapping agent (Listings 8.4 through 8.8) is a lack of generality; each swapping agent can – and is intended to – swap variable a with variable b specifically. Clarification of the conventions for giving parameters to the LSD agents is required in this situation. If LSD agent parameters are to be interpreted as call-by-value parameters, the variables in the parameter list cannot be redefined[4]. Swap(a, b) does not allow redefinition of a and b, and cannot swap the two. The swapping agent in Listing 8.9 illustrates a proposed syntax for call-by-reference parameters. Like the call-by-value parameters for LSD agents that were described by Slade in [Slade89], call-by-reference parameters serve two purposes: i) to pass information to the agent to be instantiated and ii) to disambiguate identifiers of the

---

[4] The C language has only call-by-value parameters, but it provides the & and * operators to return the address and the content of a variable respectively; this achieves the same effect as call-by-reference.

agent instances. In respect of ii), where the *values* of the call-by-value parameters are used to identify agent instances, call-by-reference parameters make it possible to identify agent instances using parameter *names*. Call-by-reference parameters also have the advantage that they can be redefined by the instantiated agent.

```
agent swap()[a,b] {
state      done[a,b] = false
oracle     a, b, done[a,b]
handle     a, b, done[a,b]
protocol
   !done[a,b] -> (a = |b| // b = |a|); done[a,b] = true
}
```

Listing 8.9: A Swapping Agent Using Call-by-Reference Parameters

8.4.4. Hidden-Text Annotation

The above suggestions have addressed those simulation issues raised by questions (Q3) and (Q5) in §8.3. The other simulation issues could be addressed by inserting *hidden-text* into an LSD specification. By *hidden-text* I mean the use of a programming interface in which the text that accompanies an LSD specification is not normally shown or editable unless specifically requested by the programmer. For example, an interface may be designed in such a way that a double-click of the mouse on an oracle variable will open a simple script, which specifies the relationship between the agent's perception of the variable and its authentic value[5]. In a similar spirit, we may associate buttons with the guarded actions, so that the frequency of guard evaluation, action responding time and the delays between actions can be recorded and modified without actually changing the LSD specification.

This way of annotation does not alter the interpretation of LSD (an LSD specification is still describing a family of behaviours) but, at the same time, provides a

---

[5] The *authentic value* of a variable is the value associated with its (unique) occurrence as an owned variable.

convenient representation for the simulation decisions required in animating a particular operational behaviour.

8.5. Translation from ADM to EDEN

8.5.1. Motivation

The current ADM language has two serious limitations:

1. The only output channel of ADM is via the print statement. This method is best suited for providing information in a procedural fashion rather than describing a state in a definitive manner. In contrast, the Scout system aims at graphical representation of state. Since Scout is an EDEN-based system, translating ADM programs into EDEN programs will greatly enhance the presentation of the ADM simulation corresponding to an LSD specification.

2. ADM has a highly limited underlying algebra. The current ADM language has only the *integer* data type. This restricts the range of LSD specifications that ADM can simulate. This restriction can be overcome if an LSD specification can be simulated by a system accepting different definitive notations. This section shows that it is possible to translate from ADM to EDEN. This means that the transformation techniques described in §8.3 can be adapted in principle to simulate LSD in the Scout system directly.

8.5.2. The Translation Scheme

An entity instance in ADM comprises two parts: the *definition* part and the *action* part. A definition in ADM can be simulated as an EDEN definition; a guarded action in ADM can be simulated by an if-statement in EDEN. The main difficulty in the translation comes from the fact that ADM performs actions and redefinitions in parallel while EDEN is basically a sequential language. ADM divides the system time into slots. In the first slot the guards are evaluated and the actions to be performed are recorded in an

action store. The actions are then performed in the second time slot and the guards are re-evaluated. The actions caused by the re-evaluation of guards are performed in the third slot and so on. On the other hand, consider the following plausible EDEN implementation of two ADM guarded actions:

```
if (guard1) { action1(); }      /* guard1 -> action1() */
if (guard2) { action2(); }      /* guard2 -> action2() */
```

action1() may change the value of guard2 that result in a false invocation or suppression of action2(). This means that the evaluation of guards and performance of actions must be separated in different time slots in order to avoid interference. The solution employed in our translator is based upon delaying the execution of actions by means of saving the actions in a *message queue* (the same communication method used between EDEN and the X window interface EX).

```
proc clocking : sysClock, stopClock {
   if (!stopClock && sysClock < stopTime) {
      if (Pause > 0)
         sleep(Pause / 2);    /* delay for Pause/2 seconds */
      if (sysClock != -1) {
         SendToEden("sysClock = -1;\n");
                  /* SendToEDEN sends an EDEN statement to EDEN
                     via a message queue */
      } else {
         nextClock++;
         if (!Silent)
           SendToEden("writeln(\"time = \", nextClock);\n");
         SendToEden("sysClock = nextClock;\n");
      }
   }
}

stopClock = 1;    /* set stopClock to stop clocking */
stopTime = 30;    /* set stopTime to the system exit time */
Silent = 0;       /* set to suppress showing of time */
Pause = 1;        /* minimum gap between two system clock pulses */
```

Listing 8.10: EDEN Simulation of a Two-Phase Clock

The use of the message queue is similar to that in the simulation of a system clock in the last chapter, except that a two-phase clock is simulated here. In Phase I, all guards are evaluated and the actions to be taken are sent to EDEN via message queue; in

Phase II, all actions in the message queue are retrieved and executed. In effect, the message queue becomes a buffer similar to the action store in ADM. The two-phase clock is simulated by the EDEN action and definitions in Listing 8.10.

When stopClock is unset (defined as 0), the clocking action will start. As a result the clocking action will be continuously invoked unless stopClock is set again or the predefined stopping time, stopTime, is reached. This is because in each invocation, clocking will generate a redefinition of sysClock, which is one of the triggering variables of the clocking action itself. The redefinition will become active only after all the redefinitions and actions in the current phase are executed. If the variable Silent is 0 (default value), a message showing the current system time will be displayed. This function is not essential when the program is executed in conjunction with the Scout system, since Scout can be used directly to display time. The Pause variable sets the minimum time between two system clock pulses. Since in between two clock cycles, there may be different number of actions taking place, setting a minimum clock rate will make the simulation run more evenly (but more slowly).

Using this two-phase clock, an ADM guarded action is translated into an EDEN action in the way illustrated by the following example. A guarded action of the station-master (sm) entity is:

```
ready && whistled && !sm_raised_flag
      print("Station master raises his flag")
            -> sm_flag = true; sm_raised_flag = true
```

Its EDEN translation is:

```
proc sm_action_6[6] : sysClock {
   if (sysClock == -1) return;
   if (isTrue(ready) && isTrue(whistled) &&
         !isTrue(sm_raised_flag)) {
      writeln(\"Station master raises his flag\");
      SendToEden(\"sm_flag is TRUE; sm_raised_flag is TRUE;\\n\");
   }
}
```

---

[6] The name sm_action_6 corresponds to the sixth action of the sm entity.

When sysClock is −1, i.e. in Phase II, the if-statement will not be executed; otherwise the guard as in the condition of the if-statement is evaluated and the possible actions will be executed in the next Phase II (sysClock == −1).

An ADM entity is translated into a group of EDEN definitions and actions. An ADM entity specification is therefore translated to an EDEN procedure which generates the corresponding definitions and EDEN actions. The instantiation of an entity is the execution of this procedure. The deletion of an entity is the removal of the definitions and actions from the EDEN interpreter (this is done by means of the forget statement of EDEN).

The whole EDEN translation of the sm entity is too long to be included in the main text. The interested reader can see Appendix H for the whole Railway Station Simulation Example in LSD, ADM and EDEN.

The current deficiency of this translation scheme is that it cannot detect conflicts, whilst the ADM translator can. Conflict detection cannot be done easily because this EDEN implementation does not have a run set equivalent to that of the ADM interpreter. The actions in the message queue should not be treated as parallel actions because the message queue is also used for communicating between EDEN and the EX graphics interface. Without a proper run set, analysis of the parallel actions cannot be performed. A better translation scheme should therefore include a proper simulation of a run set. However, the current translator has demonstrated the possibility of automatic translation from ADM to EDEN. Taking account of the suggestions for enhancing the LSD notation in §8.4, there is then a good prospect of using LSD more conveniently and of overcoming the current limitations of ADM. In this way, the LSD programming environment will become suitable for exploratory development.

9

Summary and Conclusion

This thesis aims to justify the claim that definitive programming is a good paradigm for exploratory programming. The exploratory software development method is employed when the specification of a problem is not known or unclear. Exploratory software development employs a Run-Understand-Debug-Edit (RUDE) cycle. To make exploratory software development efficient, exploratory software should be continuously executable, easily extendible, conveniently explorable and usefully explainable. Many of our previous publications encourage us to consider definitive programming as a paradigm for exploratory programming. This thesis discusses in detail the relationship between definitive programming and exploratory software development.

The Definition-based State Transition (DST) model is the essence of the definitive paradigm. The DST model is a state-transition model in which a state is represented by a set of definitions and the transitions are represented by redefinitions. From the operational point of view, a definitive state provides data dependency

information and methods of maintaining the state. Hence, definitive programs should be, in principle, highly parallelisable. From the semantic point of view, definitive programming is different from conventional programming in that a variable stores a formula rather than a concrete value. A definition denotes a value (the evaluation of the formula), gives meaning to the value (the formula) and specifies the relationships between the variables in the formula and the variable that appears in the left-hand-side of the definition. Therefore, the overall state change can be predicted when some of the variables get redefined. This means that the potential changes to the state are captured in the definition of the state itself. This makes a definitive paradigm useful for modelling applications.

Many common software tools use concepts similar to the definitive principle. This indicates that definitive programming has great potential for use in developing realistic applications. The Jugs screen layout design exercise further strengthens the belief that definitive notations are particular well-suited for design applications. Definitive programming uses domain-specific underlying algebra, allows flexible definition arrangement and integrates the design and simulation processes. All these features enable convenient modelling of states and redesigning of the models.

The relationship between the value of a variable and the values of variables on which it depends is analogous to a mechanical linkage. There is inseparable propagation of value changes within a single transition of state. Therefore, definitive notations such as Scout provide a neat way of separating the presentation of state from the definition of the state. This allows the programmer to develop the definitive state model without bothering too much about the issues concerning the realisation of states.

Each definitive notation is specifically designed for a class of applications. This is an advantage where modelling is concerned. On the other hand, it is a disadvantage with respect of general-purpose programming. To define a state, we need several

definitive notations to describe different aspects of that state. So there is a need to integrate definitive notations.

The Scout project was the response to this demand for integration. The Scout project addresses this problem in two ways: through the design and implementation of the Scout notation and through deriving a scheme for communication between definitive variables. The Scout project can be viewed as a constructive solution to the integration problem in two ways:

- While DoNaLD and ARCA concentrate on how to define a model, Scout concentrates on how to present the model. When generating a screen display is the common goal for different definitive notations, Scout can be the link between those definitive notations.

- A bridging definition is a channel through which definitive notations can communicate. This generally establishes a connection between different definitive notations independent of the assumption that screen display is the common ground.

The representation of states by sets of definitions must be complemented by some way of specifying state transitions. We have considered an agent-oriented definition-based specification language – LSD. Clearly, LSD adopts modelling as a programming strategy – it models the perception and reaction of the agents involved. This strategy matches the rich modelling property of definitions in specifying states.

In connection with exploratory software development, the modelling orientation of definitive programming surely makes a program highly explainable. This alone is not enough to justify definitive programming as exploratory programming paradigm. Other principles of exploratory software development must also be observed. These relate to:

1) how easy is it to change a specification,

2) how easy is it to transform a specification into an executable program and

3) how efficient is the execution.

In responding to these issues, we would argue that:

1) Changing a definition or an agent specification is, in principle, simple. The most difficult aspect is when a change in the specification requires an extension to the underlying algebra or even a new definitive notation. However, this thesis has shown a systematic way of implementing a definitive notation, and this scheme is proven to be simple by our experience in implementing existing definitive notations.

2) Transforming definitions in other definitive notations into executable (EDEN) definitions can be done on-line and is a fully-automatic process; transforming an LSD specification into an executable program is non-trivial. But with the improvement of notations suggested in chapter 8, hidden-text annotation and the ADM-to-EDEN translator, it is reasonable to believe that transformation of LSD into an executable program can be close to fully, if not completely, automated.

3) Many publications mentioned in this thesis have already discussed the potential for concurrency in definitive programming. On this basis, we can be confident that the execution of definitive program can be highly efficient.

Current definitive systems are far from perfect, but many suggestions in the thesis have yet to be implemented, and the evidence presented in this thesis is sufficient to justify the assertion that "definitive programming is a good paradigm for exploratory programming".

References

[Adobe85] Adobe Systems Inc. "PostScript Language Reference Manual", Addison-Wesley, 1985

[Aonuma88] Anonuma, Tatsuo. "An Interactive Simulation Modelling System: DYNAGRAPH for Multi-Period Planning on an APL Spreadsheet", APL88, 10-18, 1988

[BABH86] Beynon, W M, Angier, D, Bissell, T, Hung, S. "DoNaLD: a Line-Drawing System based on Definitive Principles", University of Warwick Computer Science Research Report 86, October 1986

[Backus78] Backus, J. "Can Programming be Liberated from the von Neumann Style?", Communication of the ACM, Vol. 21, No. 8, 1978, pp. 613-641

[Baldwin87] Baldwin, D. "Why We Can't Program Multiprocessors the Way We're Trying to Do It Now", Technical Report 224, Department of Computer Science, University of Rochester, August 1987

[BBY92] Beynon, W M, Bridge, I, Yung, Y P. "Agent-oriented Modelling for a Vehicle Cruise Controller", in Proc ASME conference on Engineering Systems Design and Analysis, Turkey, June-July 1992, pp. 159-165

[BCMZ88] Brown, D W, Carson, C D, Montgomery, W A, Zislis, P M. "Software Specification and Prototyping Technologies", AT&T Technical Journal, July/August 1988, pp.33-45

[BCRY90] Beynon, M, Cartwright, A J, Russ, S B, Yung, Y P. "Programming Paradigms and the Semantics of Geometric Symbols, in Proc. W/S "Visual Interfaces to Geometry" in conjunction with CHI'90, Seattle, April 1990

[BD86] Borning, A, Duisberg, R. "Constraint-Based Tools for Building User Interfaces", ACM Transactions on Graphics, Vol. 5, No. 4, Oct 1986, pp 345-374

[BDMN79] Birtwistle, G M, Dahl, O J, Myhrhaug, B, Nygaard, K. "Simula Begin", Chartwell-Bratt Ltd., 1979

[Beynon85] Beynon, W M. "Definitive Notations for Interaction", Proc. BCS Conference "People and Computers: Designing the Interface", ed. Johnson and Cook, CUP, 1985

[Beynon86a] Beynon, W M. "ARCA: a Notation for Displaying and Manipulating Combinatorial Diagrams", Univerity of Warwick Computer Science Research Report 78, July 1986

[Beynon86b] Beynon, W M. "The LSD Notation for Communicating Systems", University of Warwick Computer Science Research Report 87, November 1986

[Beynon88a] Beynon, W M. "Definitive Principles for Interactive Graphics", NATO ASI Series F: 140, 1988

[Beynon88b] Beynon, W M. "Definitive Programming for Parallelism", University of Warwick Computer Science Research Report 132, October 1988

[Beynon89] Beynon, W M. "Evaluating Definitive Principles for Interaction in Graphics", in Proc. of CG International '89, "New Advances in Computer Graphics", Springer-Verlay, 1989

[Beynon92] Beynon, W M. "Programming Principles for the Semantics of the Semantics of Programs", Research Report 205, Computer Science Department, University of Warwick, 1992

[BF85] Burden, R L, Faires, J D. "Numerical Analysis", third edition, PWS Publishers, 1985

[BKKMSZ86] Bobrow, D, Kahn, K, Kiczales, G, Masiuter, L, Stefik, M, Zdybel, F. "Common Loops: Merging Lisp and Object Oriented Programming", in Proc. OOPSLA'86. ACM SIGPLAN 21, 11 (Nov.), 1986, pp. 17–29

[BN87] Beynon, W M, Norris, M T. "Comparison of SDL and LSD", in Proc SDL'87, ed R Saracco, P A J Tilanus, North-Holland 1987, pp. 201-209

[BNRSYY89] Beynon, W M, Norris, M T, Russ, S B, Slade, M D, Yung, Y P, Yung, Y W. "Software Construction Using Definitions: an Illustrative Example", University of Warwick Computer Science Research Report #147

[BNS88] Beynon, W M, Norris, M T, Slade, M D. "Definitions for Modelling and Simulating Concurrent Systems", Proceedings IASTED Conference ASM 1988, Acta Press, 1988

[Bobrow85] Bobrow, D G. "If Prolog is the Answer, What is the Question? Or What It Takes to Support AI Programming Paradigms", IEEE Transactions on Software Engineering, Vol SE-11, No. 11, Nov. 1985, pp. 1401-1408

[Booch91] Booch, G. "Object-Oriented Design with Applications", The Benjamin/Cummings Publishing Company, Inc, 1991

[BR89] Beynon, W M, Russ, S B. "The Development and Use of Variables in Mathematics and Computer Science", in Proc. IMA conference "The Mathematical Revolution Inspired by Computing", April 1989

[BR92] Beynon, W M, Russ, S B. "The Interpretation of States: a New Foundation for Computation?", Research Report 207, Computer Science Department, University of Warwick, 1992

[Brooks86] Brooks, F P. "No Silver Bullet: Essence and Accidents of Software Engineering", Computer, Vol. 20, No. 4, 1987, pp. 10-19

[Brooks91] Brooks, K P. "Lilac: A Two-View Document Editor", Computer, 7-19, June 1990

[BRSYY89] Beynon, W M, Russ, S B, Slade, M D, Yung, Y P, Yung, Y W. "Definitive Principles and the Specification of Software", University of Warwick Computer Science Research Report #146

[BRY90] Beynon, W M, Russ, S B, Yung, Y P. "Programming as Modelling: New Concepts and Techniques", in Proc ISLIP'90, Computing & Info Sci Dept, Queen's University, Canada 1990

[BS81] Bobrow, D G, Stefik, M. "The Loops Manual", Technical Report KB-VLSI-81-13. Knowledge Systems Area, Xerox Palo Alto Research Center, 1981

[BSY88] Beynon, W M, Slade, M D, Yung, Y W. "Parallel Computation in Definitive Models", in Proc CONPAR'88, British Computer Society Workshop Series, CUP 1989, pp. 359-367

[BSY90] Beynon, W M, Slade, M D, Yung, S. "Protocol Specification in Concurrent Systems Software Development", University of Warwick Computer Science Research Report #163

[BY90] Beynon, W M, Yung, Y P. "Definitive Interfaces as a Visualisation Mechanism", in Proc. Graphics Interface '90, Canadian Information Processing Society, 1990, pp. 285–292

[BY92] Beynon, M, Yung, Y P. "Agent-oriented Modelling for Discrete-Event Systems", in Proc IEE Colloquium "Discrete-Event Dynamic Systems", June 1992

[BYAB91] Beynon, W M, Yung Y P, Atkinson, M D, Bird, S R. "Programming Principles for Visualization in Mathematical Research", in Proc. Computgraphics'91, Sesimbra, Portugal, Sept 1991

[CB92] Cartwright, A J, Beynon, W M. "Enhancing Interaction in Computer-Aided Conceptual Design", in Proc ICMA'92, Hong Kong, August 1992

[CFV88] Cugini, U, Folini, F, Vicini, I. "A Procedural System for the Definition and Storage of Technical Drawings in Parametric Form", in Eurographics '88, North-Holland, 1988, pp. 183-196

[Chan89] Chan, S. "Enhancing the DoNaLD Line Drawing System", final year project report, Deparment of Computer Science, University of Warwick, 1989

[Clocksin87] Clocksin, W F. "Principles of the DelPhi Parallel Inference Machine", The Computer Journal, Vol. 30, No. 5, 1987, pp. 386–392

[Cox84] Cox, B J. "Message/Object Programming: An Evolutionary Change in Programming Technology", IEEE Software, vol. 1, no. 1, January 1984, pp. 50–69

[Cox86] Cox, B J. "Object-Oriented Programming: An Evolutionary Approach", Addison-Wesley, Reading (Mass.), 1986

[CP87] Chan, W T, Paulson B C. "Exploratory Design Using Constraints", AI EDAM, 1987 1(1), pp. 59-71

[CW89] Chmilar, M, Wyvill, B. "A Software Architecture for Integrated Modelling and Animation", in "New Advances in Computer Graphics", Proc. of CG International'89, Springer-Verleg, 1989, pp. 257–276

[Deutsch89] Deutsch, M S. "Enhancing Testability with Scenario-Oriented Engineering", in Proc 6th Int Conf on Testing Computer Software: Managing the Testing Process, May 1989

[DN66] Dahl, O J, Nygaard, K. "SIMULA—an Algol-based Simulation Language", Communications of the ACM. 9. 1966, pp. 671–678

[DT88] Danforth, S, Tomlinson, C. "Type Theories and Object-Oriented Programming", ACM Computing Surveys, Vol. 20, No. 1, March 1988, pp. 29-72

[FB83] Fischer, G, Boecker, H, D. "The Nature of Design Processes and How Computer Systems Can Support Them", in Integrated Interactive Computing Systems, ed. P Degano and E Sandewall, North Holland, 1983, pp. 73-86

[Foley87] Foley, J. "Models and Tools for the Designers of User-Computer Interfaces", Teport GWU-IIST-87-03, Department of EE & CS, George Washington University, March 1987

[FP86] Fuller, N, Prusinkiewicz, P. "L.E.G.O. – An Interactive Graphics System for Teaching Geometry and Computer Graphics", in Proc CIPS Edmonton'86, pp. 75-84

[FP88] Fuller, N, Prusinkiewicz, P. "Geometric Modelling with Euclidean Constructions", in "New Trends in Computer Graphics", Springer-Verlag, 1988, pp. 379–391

[FP89] Fuller, N, Prusinkiewicz, P. "Applications of Euclidean constructions to computer graphics", The Visual Computer (1989) 5, Springer-Verlag, 1989, pp. 63-67

[GHT84] Glaser, H, Hankin, C, Till, D. "Principles of Functional Programming", Prentice-Hall International, 1984

[GO90] Gilly, D, O'Reilly, T. "The Definitive Guides to the X Window System", O'Reilly & Associates Inc., 1990

[GR83] Goldberg, A, Robson, D. "Smalltalk-80: The Language and Its Implementation", Addision Wesley, 1983

[Gray89] Gray, P D. "Interactive User Interface Design: the Chimera UIMS", Digest No: 1989/135, Colloquium on "User Interface Management Systems", Computing and Control Division, IEE, 1989, pp. 6/1–6/5

[Green89] Green, T R G. "Cognitive Dimensions of Notations", in Proc HCI'89, 443-460, 1989

[Harel92] Harel, D. "Biting the Silver Bullet", Computer, Jan. 1992, pp. 8-20

[Harland88] Harland, D M. "REKURSIV Object-Oriented Computer Architecture", Ellis Horwood Ltd, August 1988

[Harper86] Harper, R. "Introduction to Standard ML", LFCS Report ECS-LFCS-86-14 (also published as CSR-220-86), LFCS, Department of Computer Science, University of Edinburgh, Nov. 1986

[Hewett89] Hewett, Thomas T. "Towards a Rapid Prototyping Environment for Interface Design: Desirable Features Suggested by The Electronic Spreadsheet", in Proc HCI'89, 305-314, 1989

[HH89] Hartson, H R, Hix, D. "Human-Computer Interface Development: Concepts and Systems for Its Management", ACM Computer Surveys, Vol. 21, No. 1, March 1989, pp. 5-92

[Hill92] Hill, R D. "The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications", in Proc CHI'92, 1992, pp. 335-342

[HS78] Hillis, W, Steele, G. "Data Parallel Algorithms", Communications of the ACM, August 1978 (21:8), pp. 666–677

[Hudak89] Hudak, P. "Conception, Evolution, and Application of Functional Programming Languages", ACM Computing Surveys, Vol. 21, No. 3, September 1989, pp. 359–411

[Hui90] Hui, F. "Translation of Definitive Notation to C Code", final year project report, Department of Computer Science, University of Warwick, 1990

[Iannucci88] Iannucci, R A. "A Dataflow / von Neumann Hybrid Architecture", Tech. Report MIT/LCS/TR-418, Laboratory for Computer Science, MIT, 1988

[ITT67] "APT part-programming", ITT Research Institute, McGraw-Hill, 1967

[Iu89] Iu, K F. "An Interface for DoNaLD in a Windowing Environment", final year project report, Department of Computer Science, University of Warwick, 1989

[Jackson83] Jackson, M. "System Development", Prentice-Hall International, 1983

[Jackson88] Jackson, Mary. "Advanced Spreadsheet Modelling with Lotus 1-2-3", John Wiley & Sons, 1988.

[JB88] Johnson, Jeff; Beach, Richard J. "Styles in Document Editing Systems", Computer, 32-43, January 1988

[Jeet87] Jeet, E J. "A Relationship-Based Interactive Graphical Diagram Editor", PhD thesis, Department of Computer Science, University of Kent, Nov. 1987

[Jorrand87] Jorrand, P. "Design and Implementation of a Parallel Inference Machine for First Order Logic: An Overview", Proc. PARLE 87, volumn 1: Parallel Architectures, 1987, pp. 434-445

[Kay84] Kay, Alan. "Computer Software", in Scientific American, Vol 251, No 3, 41-47, Sept 1984

[KKW87] Kunz, J C, Kehler, T P, Williams, M D. "Applications Development Using a Hybrid AI Development System", in Artificial Intelligence Programming Environments, ed. Robert Hawley, Ellis Horwood Limited, 1987. pp. 177-196

[Knuth84] Knuth, D E. "The Texbook", Addison-Wesley, 1984

[Kokol88] Kokol, Peter. "Spreadsheet Language Level: How High Is IT?", SIGPLAN Notices, vol 23, No 6, 121-134, June 1988

[Landin66] Landin, P J. "The Next 700 Programming Languages", Communications of the ACM, Vol. 9, No. 3, March 1966, pp. 157-166

[Lewis87] Lewis, C. "Using the NoPumpG Primitive", Department of Computer Science and Institute of Cognitive Science, University of Boulder, 1987

[Lieberman86] Lieberman, H. "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", Proc First ACM Conf on Object-Oriented Programming Systems, Languages and Applications, SIGPLAN Notices, vol 21, no 9, 1986, pp. 214-223

[LP86] Lang, K, Perlmutter, B. "OakLisp: An Object-Oriented Scheme with First Class Types", OOPSLA 86. ACM SIGPLAN 21, 11 (Nov.), 1986, pp. 30–37

[Lucassen87] Lucassen, J M. "Types and Effects, Towards the Integration of Functional and Imperative Programming", PhD thesis, Laboratory for Computer Science, MIT, Aug. 1987

[LV91] Laffra, C, Van Den Bos, J. "Constraints in Concurrent Object-Oriented Environments", in OOPS Messenger Vol 2 No. 2, April 1991, pp.64-67

[Martin87] Martin, M M. "Tutorial Introduction to ww for C", Rutherford Appleton Laboratory, 1987

[MAS88] McCormack, J, Asente, P, Swick, R R. "X Toolkit Intrinsics – C Language Interface", Digital Equipment Corp., March 1988

[MBF89] Maloney, J H, Borning, A, Freeman-Benson, B N. "Constraint Technology for User Interface Construction in ThingLab II", in Proc. OOPSLA'89, Oct 1989, pp. 381-388

[Meads87] Meads, J. "The Standards Factor", SIGCHI Bull., Vol. 19, No. 1, July 1987, pp. 34-35

[Meyer88] Meyer, B. "Object-Oriented Software Construction", Prentice-Hall International, 1988

[MGJ84] McCrosky, C D, Glasgow, J I, Jenkins, M A. "Nial: A Candidate Language for Fifth Generation Computer Systems", Technical Report 84-159, Departmentt of Computing and Information Science, Queen's University at Kingston, Ontario, Canada

[Michael89] Michael, D. "Trident – Translator of Definitive Notations", final year project report, Department of Computer Science, University of Warwick, 1989

[Musciano88] Musciano, C. "Tooltool User's Guide - Version 2.0", Advanced Technology Department, Harris Corporation, 1988

[Myers89] Myers, B A. "The State of the Art in Visual Programming and Program Visualization", in Graphics Tools for Software Engineers, 3-26, 1989

[Myers90] Myers, B A. "A New Model for Handling Input", ACM Transactions on Information Systems, Vol 8, No. 3, July 1990, pp. 289-320

[Naur63] Naur, P. Ed. "The Revised Report on the Algorithmic Language ALGOL 60", Communication of the ACM, Jan 1963, pp. 1-17

[Nelson90] Nelson, M L. "An Introduction to Object-Oriented Programming", Naval Postgraduate School, Monterey, CA, Technical Report No 52-90-024, Apr 1990

[Nelson91] Nelson, M L. "An Object-Oriented Tower of Babel", OOPS Messenger, Vol. 2, No. 3, July 1991, pp.3-11

[Parsons91] Parsons, J. "Enhancement of the DoNaLD Translator", final year project report, Deparment of Computer Science, University of Warwick, 1991

[Partridge86] Partridge, D. "Artificial Intelligence: Applications in the Future of Software Engineering", Ellis Horwood, 1986

[Pemberton92] Pemberton, S. "Programming Aspects of Views, an Open-Architecture Application Environment", in poster presentation, HCI'92, York, Sept 1992

[PW87] Partridge, D, Wilks, Y. "Does AI Have a Methodology Which is Different from Software Engineering?" Artificial Intelligence Review 1 (1), 1987, pp. 111-120

[PW88] Petre, M, Winder, R. "On Languages, Models and Programming Styles", The Computer Journal, Vol 33, No. 2, pp. 193-180

[Ramsay87] Ramsay, A. "Embedding very high level languages", in Artificial Intelligence Programming Environments, ed. Robert Hawley, Ellis Horwood Limited, 1987. pp. 61-78

[Research87] Research Software Ltd. "Miranda System Manual", 1987

[SB86] Stefik, M, Bobrow, D G. "Object-Oriented Programming: Themes and Variations", AI Mag. 6, 4, 1986, pp. 40-62

[Sheil83] Sheil, B. "Power Tools for Programmers", Datamation, Feb 1983, pp. 131-144

[Slade89] Slade, M D. "Definitive Parallel Programming", MSc. thesis, Department of Computer Science, University of Warwick, 1989

[SM88] Myers, B A, Szekely, P A. "A User Interface Toolkit Based on Graphical Objects and Constraints", in Proc OOPSLA '88, 1988, pp. 36-45

[Smillie88] Smillie, K W. "Array Theory and the NIAL Programming Language", BIT 28 (1988), pp. 439–449

[Smith87] Smith, B C. "Two Lessons of Logic", Comput Intell 3, 1987, pp. 151-160

[Snyder87] Snyder, A. "Inheritance and the Development of Encapsulated Software Components", in Proceedings of the 20th Hawaiian International Conference on Systems Sciences. Software Track, Western Periodicals, North Hollywood, Calif., 1987, pp. 227–238

[Sommerville89] Sommerville, I. "Software Engineering", 3rd edition, Addison-Wesley, 1989

[Sowa87] Sowa, M. "A Method for Speeding up Serial Processing in Dataflow Computers by Means of a Program Counter", The Computer Journal, Vol. 30, No. 4, 1987, pp. 289–294

[Stidwill89] Stidwill, J M. "The CADNO Programming Language", third year project report, Department of Computer Science, University of Warwick, !989

[Streitz88] Streitz, N A. "Writing is Rewriting: A Cognitive Framework for Computer-Aided Authoring", in Proc 4th European Conference on Cognitive Ergonomics, Cambridge, 1988

[Stroustrup84] Stroustrup, B. "Data Abstraction in C", AT&T Bell Laboratories Technical Journal, vol. 63, no. 8, Part 2, October 1984, pp. 1701–1732

[Stroustrup86] Stroustrup, B. "The C++ Programming Language", Addison Wesley, 1986

[Sun84] Sun Mircosystems Inc. "Sun Windows Programmers' Guide", Mountain View, Calif., 1984

[Sun87] Sun Mircosystems Inc. "NeWS Manual", Mountain View, Calif., 1987

[Thimbleby90] Thimbleby, H. "User-Interface Design", Addison-Wesley, 1990

[Trenouth91] Trenouth, J. "A Survey of Exploratory Software Development", The Computer Journal, Vol. 34, No. 2, 1991, pp. 153-163

[Turner81] Turner, D A. "Recursion Equations as a Programming Language", in "Functional Programming and its Applications", edited by Darlington J, Henderson P and Turner D A, Cambridge University Press, 1981, pp. 1-28

[Turner83] Turner, D A. "The Future of Programming Languages", Infotech state of the ART report, 1983, pp. 129-135

[Turner85] Turner, D A. "Miranda: a Non-strict Functional Language With Polymorphic Types", in Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture. Springer Verlag Notes in Computer Science – 201, Nancy, France, September 1985.

[Turner86] Turner, D A. "An Overview of Miranda". SIGPLAN Notices, December 1986

[Ullman82] Ullman, J D. "Principles of Database Systems", Computer Science Press, Rockville Maryland, second edition, 1982

[van Denneheuvel91] van Denneheuvel, S. "Constraint-solving on database systems: design and implementation of the rule language RL/1", CWI Amsterdam 1991

[Veen86] Veen, A H. "Dataflow Machine Architecture", ACM Computer Surveys, Vol. 18, No. 4, December 1986.

[VLM88] Vaucher, J, Lapalme, G, Malenfant, J. "SCOOP: Structured Concurrent Object Oriented Prolog", in Proc. ECOOP'88, August 1988, pp. 191-135

[Wadge85] Wadge, W W. "VISCID, A Vi-like Screen Editor Written in LUCID", Department of Computer Science, University of Victoria, Canada, June 1985

[Wegner87] Wegner, P. "Dimensions of Object-Based Language Design", Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87) Conference Proceedings, Oct 1987, special issue of SIGPLAN Notices, Vol 22, No 12, Dec 1987, pp. 168-182

[Whitefield89] Whitefield, A. "Constructing Appropriate Models of Computer Users: The Case of Engineering Designers", in Cognitive Ergonomics and Human-Computer Interaction, edited by J Long and A Whitefield, Cambridge, 1989, pp. 66-94

[WP88] Winder, R L, Pun, W W Y. "Towards a Design Method for Object-Oriented Programming", Department of Computer Science Research Note RN/88/1, University College London, Jan. 1988

[Wyvill75] Wyvill, B. "An Interactive Graphics Language", PhD thesis, University of Bradford, 1975

[Yung87] Yung, Y W. "EDEN: Evaluator of Definitive Notations", final year undergraduate project report, Department of Computer Science, University of Warwick, 1987

[Yung88] Yung, Y P. "Design and Implementation of SCOUT – A Definitive Notation for Describing SCreen LayOUT", final year undergraduate project report, Department of Computer Science, University of Warwick, 1988

[Yung89] Yung, Y W. "EDEN: An Engine for Definitive Notations. Design, Implementation and Evaluation", MSc. thesis, Department of Computer Science, University of Warwick, 1989

Appendix A

Technical Document of the Scout System

Technical Document for the Scout System

Technical Document for the Scout System

1. Structure of This Document

There are three terms relating to 'Scout': the Scout notation, the Scout-to-EDEN translator and the Scout system. In this document, Section 2 gives the summary of the Scout notation. Section 3 is the user manual of the Scout-to-EDEN translator. Section 4 is the user manual for EX – an EDEN/X Window interfacing program used in the Scout system. Section 5 describes the Scout system. The Scout system provides an integrated programming environment for the Scout notation, the DoNaLD notation and the EDEN language. As a result of the integration, some modifications have been made to the original DoNaLD system implemented by Edward Yung in 1988 and the EDEN interpreter as documented in [5]. Section 6 explains the recent changes to these definitive systems.

2. The Scout notation

The Scout notation is a definitive notation for describing screen layout. To understand what a definitive notation is, see [1,2,3]. To understand the concept of the Scout notation, see chapter 4 of [4]. The following is the BNF of the Scout notation.

<statement> :: <declaration> | <definition>

<declaration>[1] :: <type_name> <var_list> ';'

<type_name> :: 'string' | 'integer' | 'point' | 'box' | 'frame' | 'window' | 'display'

<var_list> :: <var> | <var_list> ',' <var>

<var>[2] :: <string_var> | <integer_var> | <point_var> | <box_var> | <frame_var>
| <window_var> | <display_var>

<definition> :: { 'string' | '' } <string_var> '=' <string_exp> ';'
| { 'integer' | '' } <integer_var> '=' <integer_exp> ';'
| { 'point' | '' } <point_var> '=' <point_exp> ';'
| { 'box' | '' } <box_var> '=' <box_exp> ';'
| { 'frame' | '' } <frame_var> '=' <frame_exp> ';'

---

[1] A variable once declared cannot be redeclared to other data types.

[2] All variable names are strings of alphanumeric starting with a letter.

| { 'window' | '' } <window_var> '=' <window_exp> ';'
| { 'display' | '' } <display_var> '=' <display_exp> ';'

<string_exp> :: <string>
| <string_var>
| <string_exp> '//' <string_exp>
| 'strcat' '(' <string_exp> ',' <string_exp> ')'
| 'substr' '(' <string_exp> ',' <integer_exp> ',' <integer_exp> ')'
| 'itos' '(' <integer_exp> ')'
| <window_exp> '.' 'string'
| <window_exp> '.' 'pict'
| <window_exp> '.' 'bgcolor'
| <window_exp> '.' 'fgcolor'
| 'if' <integer_exp> 'then' <string_exp> 'else' <string_exp> 'endif'

<integer_exp> :: <integer>
| <integer_var>
| <integer_exp> '.' 'c'
| <integer_exp> '.' 'r'
| <integer_exp> <int_op> <integer_exp>
| '-' <integer_exp>
| '(' <integer_exp> ')'
| 'strlen' '(' <string_exp> ')'
| <point_exp> '.' <integer_exp>
| <window_exp> '.' 'xmin'
| <window_exp> '.' 'ymin'
| <window_exp> '.' 'xmax'
| <window_exp> '.' 'ymax'
| 'if' <integer_exp> 'then' <integer_exp> 'else' <integer_exp> 'endif'

<int_op> :: '+' | '-' | '*' | '/' | '%' | '==' | '!=' | '>' | '>=' | '<' | '<=' | '&&' | '||'

<point_exp> :: <point_var>
| '{' <integer_exp> ',' <integer_exp> '}'
| <point_exp> '+' <point_exp>
| <point_exp> '-' <point_exp>
| <box_exp> '.' <direction>
| 'if' <integer_exp> 'then' <point_exp> 'else' <point_exp> 'endif'

<direction> :: 'n' | 'e' | 's' | 'w' | 'ne' | 'nw' | 'se' | 'sw'

<box_exp> :: <box_var>
| '[' <point_exp> ',' <point_exp> ']'
| '[' <point_exp> ',' <integer_exp> ',' <integer_exp> ']'
| <frame_exp> '.' <integer_exp>
| <window_exp> '.' 'box'
| 'shift' '(' <box_exp> ',' <integer_exp> ',' <integer_exp> ')'
| 'intersect' '(' <box_exp> ',' <box_exp> ')'
| 'centre' '(' <box_exp> ',' <box_exp> ')'
| 'enclose' '(' <box_exp> ',' <box_exp> ')'
| 'reduce' '(' <box_exp> ',' <box_exp> ')'
| 'if' <integer_exp> 'then' <box_exp> 'else' <box_exp> 'endif'

<box_list> :: <box_exp> | <box_list> ',' <box_exp>

<frame_exp> :: <frame_var>
| '(' <box_list> ')'
| <window_exp> '.' 'frame'
| 'append' '(' <frame_exp> ',' <integer_exp> ',' <box_exp> ')'
| 'delete' '(' <frame_exp> ',' <integer_exp> ')'
| <frame_exp> '&' <frame_exp>
| 'if' <integer_exp> 'then' <frame_exp> 'else' <frame_exp> 'endif'

<window_exp> :: <window_var>
| '{' <window_field_list> '}'
| <display_exp> '.' <integer_exp>
| 'if' <integer_exp> 'then' <window_exp> 'else' <window_exp>
'endif'

<window_field_list> :: <window_field> | <window_field_list> ',' <window_field>

<window_field> :: 'type' ':' { 'TEXT' | 'DONALD' | 'ARCA' }
| 'frame' ':' <frame_exp>
| 'string' ':' <string_exp>
| 'box' ':' <box_exp>
| 'pict' ':' <string_exp>
| 'xmin' ':' <integer_exp>
| 'ymin' ':' <integer_exp>
| 'xmax' ':' <integer_exp>
| 'ymax' ':' <integer_exp>

| { 'bgcolour' | 'bgcolor' } ':' <string_exp>
| { 'fgcolour' | 'fgcolor' } ':' <string_exp>
| 'border' ':' <integer_exp>
| 'alignment' ':' <justification>
| 'sensitive' ':' { 'ON' | 'OFF' }

<justification> :: 'NOADJ' | 'LEFT' | 'RIGHT' | 'EXPAND' | 'CENTRE'

<window_list> :: <window_exp> | <window_list> '/' <window_exp>

<display_exp> :: <display_var>
| '<' <window_list> '>'
| 'append' '(' <display_exp ',' <integer_exp> ',' <window_exp> ')'
| 'delete' '(' <display_exp ',' <integer_exp> ')'
| 'if' <integer_exp> 'then' <display_exp> 'else' <display_exp> 'endif'

Understanding the Scout Notation

Scout describes a display as (potentially) overlapping windows. For example, if display `disp` is defined as

```
disp = < win1 / win2 >
```

this means that display `disp` consists of two windows `win1` and `win2`, should `win1` and `win2` overlap, `win1` overlays `win2`.

The best way of understanding what a Scout window is is through the formula below:

$$\text{window} = \text{region} \times \text{content} \times \text{attributes}$$

A window defines a region in which something will be displayed in a certain way. There are three kinds of windows so far in the existing Scout notation: text window, DoNaLD window and ARCA window. Because of the different nature of the windows, their definitions of region, content and attributes may differ.

For a text window,

region (called a *frame*) = list of boxes

The string is filled into the first box, the remaining characters are filled into the second box and so on.

content = a character string

attribute = { fgcolour[3] | bgcolour | border | alignment }

These attributes indicate the colour of the text string, the colour of the background, whether the boxes have borders and the alignment of strings in relation to the boxes respectively.

For a DoNaLD or ARCA window,

region = a box

content = a drawing (name of the drawing)

attribute = { xmin, ymin, xmax, ymax, fgcolour, bgcolour, border }

xmin, *ymin*, *xmax*, *ymax* defines the coordinate system of the drawing; *fgcolour* and *bgcolour* defines the foreground and background colour and *border* determines whether to draw borders of the box.

The sensitive attribute is common to all three types of windows. It is used to declare that a window is sensitive to mouse and keypress actions. When this attribute is ON, a mouse action or a keypress action within the region of this window will cause a definition to be generated. If a mouse action occurs in a window and it is a DoNaLD or ARCA window, then the window name concatenated with "_mouse" will be the name of the variable to be defined; if it occurs in a text window, the window name concatenated with "_mouse_" followed by the box number will be the variable name. The value assigned to the appropriate variable records the nature and the location of the mouse action. It is a 5-tuple of (*button*, *type*, *state*, *x*, *y*) where

button = the button number pressed or released;

type = the button action (4 = pressed, 5 = released);

state = the state before the button action occurred (shift (+1), caplock (+2), control (+4), meta (+8) and was-pressed (+256)). For example, if a button is released while the shift and control keys are depressing, state will be 1 + 4 + 256 = 261;

x, *y* = the x- and y- coordinates of the mouse in the coordinate system of the window in which the mouse action occurred.

---

[3] The names `fgcolor` and `bgcolor` are synonymous of `fgcolour` and `bgcolour`.

As with mouse events, a stroke on the keyboard will generate a definition. Instead of "_mouse" or "_mouse_" followed by a box number, the variable name of the generated definition will end with "_key" or "_key_" followed by a box number. The value defined will also be a 5-tuple: (key, type, state, x, y), where key is the ascii code of the key pressed.

In principle, there could be many types of windows, many more attributes and many ways of defining regions. The current notation only demonstrates the principle of using definitions in describing screen layout.

3. The Scout-to-EDEN Translator

Synopsis

```
scout.trans [-l] filename ...
```

Options

-l Keep log in the file named `scout.log` in the working directory.

Description

`scout.trans` translates the files listed and then followed by the standard input.

`scout.trans` translates only those lines bounded by the line beginning with %scout and the line beginning with % but not followed by the word scout. For example:

...	Lines not to be translated by `scout.trans`
%scout	%scout must be put at the beginning of the line
...	Lines in the Scout notation
%other	
...	Lines in the other definitive notation

For debugging purposes, an interrogation command (?) is available.

? *variable*; Display the definition and the data type of the Scout variable.
? all; Display the definitions and the data types of all the Scout variables.

`screen` is a pre-declared variable of type `display`; It corresponds to a physical window in the X Window system.

A definition in Scout Notation will be translated to a definition in EDEN. Also the variable names are not changed after translation.

Files

See the 'Files' section in Section 4, 'The Scout System'.

4. EX – An EDEN/X Window Interface Program

Synopsis

EX *msgqid*

Description

Message queue is one of the UNIX System V's Inter-Processes Communication (IPC) methods. Every entry in a message queue consists of an integer defining the type of the message and the actual message (a string of characters). EX interprets every message of type 2 in the message queue *msgqid* as a line of command. These commands may create or destroy a window, draw things in a window or query some information of a window. Should information be passed back, a type 1 message is sent by EX to the same message queue.

EX commands

In the following commands, all strings should be quoted ("...") except for display-name, box-name and attribute name.

OpenDisplay *display-name x y width, height*
opens an new X-window with initial size *width* x *height* and location (*x*, *y*). This window is identified with the name *display-name*.

DestroyDisplay *display-name*
destroys the X-window named *display-name*.

MapDisplay *display-name*
shows the X-window named *display-name* if it were unmapped.

UnmapDisplay *display-name*
hides the X-window named *display-name*. The content of the window is retained. MapDisplay can be used to show the window again.

RestackDisplay *display-name*

restacks the sub-windows in the X-window named *display-name*.

RaiseBox *display-name box-name*

raises the sub-window named *box-name* up to the top of the window named *display-name*.

LowerBox *display-name box-name*

lowers the sub-window named *box-name* to the bottom of the window named *display-name*.

AddBox *display-name box-name x y width height*

creates a sub-window named *box-name* of size *width* x *height* in location (*x*, *y*) relative to the origin of the window named *display-name*.

ChangeBox *display-name box-name { attribute-name value }* End

changes the attributes of the sub-window named box-name to the values given. The attributes are:

internalX	the x-coordinate of the box
internalY	the y-coordinate of the box
width	the width of the box
height	the height of the box
string	the string to fit in the box
firstLineIndent	the first line indentation of the string
indent	the indentation of the rest of the lines
justify alignment	0 - no, 1 - left, 2 - centre, 3 - right, 4 - left & right
font	the font name of the string
background	background colour name
foreground	foreground colour name
border	border colour name
borderWidth	width of the border
xmin ymin xmax ymax	xmin, ymin, xmax, ymax define the coordinate system of the graph shown inside the box

MapBox *display-name box-name*
UnmapBox *display-name box-name*
DestroyBox *display-name box-name*
have similar meaning to MapDisplay, UnmapDisplay and DestroyDisplay.

StringRemain *display-name box-name*
sends a query to EX about the part of the string which is not shown in the box. EX will then send a type 1 message to the message queue and the content of the message will be the part of the string that is not displayed.

Fontwidth *display-name font*
Fontheight *display-name font*
sends a query to EX about the font size of *font*. The font name is a string. EX will then send a type 1 message to the message queue and the content of the message will be the width or the height of the font (as a string of digits) whichever appropriate.

DisplayDepth *display-name*
sends a query to EX about the depth of the display (e.g. =1 for monochromo workstations and =8 for NCD colour X-terminals). EX will then send a type 1 message to the message queue and the content of the message will be the depth of the X Window display unit (as a string of digits) whichever appropriate.

Disp2PS *display-name PS-filename*
generates a PostScript version of the current display in a file in the current directory. Note that *PS-filename* should be an unquoted string and without special characters.

: *display-name box-name line-drawing-command*
performs the line drawing command in the specified box.

line-drawing-command may be:

.w *xmin xmax ymin ymax*
Here *xmin*, *xmax*, *ymin* and *ymax* are floating point numbers.

".w" defines the coordinate mapping within the box: the bottom-left corner corresponds to the coordinate (*xmin*, *ymin*) and the top-right corner will be (*xmax*, *ymax*). Default: .w 0 1000 0 1000

.A *id attributes*

.A * *attributes*

Where id is the segment id (integer) and attributes is a comma separated entries which have the form: *attr=value*

The *attr* currently supported are `color`, `linewidth`, `linestyle` and `dash`.

The value of `color` can be any colour name recognised by the X Window system. The default colour is `black`.

The value of `linewidth` is the number of pixels. 0 (default) means 1 pixel but it executes more efficiently than linewidth 1.

The value of `linestyle` can be `solid` (default), `dotted` or `dashed`.

The value of `dash` is a string of digits which specifies the odd-even dot width. Default: 4 (the same as 44: which means 4 pixels in foreground colour followed by 4 pixels in background colour in the case of dashed line; in the case of dotted line, which means 4 pixels in foreground colour and then with the next 4 pixels untouched).

If * is used in place of *id*, the attributes affect ALL segments. Each segment's attribute settings override the global settings. Each segment has its own settings and may have more than one. The effects of attributes are incremental.

.P *id x y*

Here *id* is the segment ID (an integer) and *x* & *y* are floating points.

".P" appends a line to the specified segment.

The coordinates used here are expressed in the world coordinate system local to the box. It is also the case for the coordinates in other line drawing commands.

x and *y* determine the location of the point.

.L *id x1 y1 x2 y2*

Here *id* is the segment ID (an integer) and *x1*, *y1*, *x2*, *y2* are floating points.

".L" appends a line to the specified segment.

(*x1*, *y2*) and (*x2*, *y2*) are the two endpoints of the line.

.C *id x y radius*

Here *id* is the segment ID (an integer) and *x*, *y*, *radius* are floating points.

".C" appends a circle to the specified segment.

(*x*, *y*) is the centre of the circle and *radius* is its radius.

.E *id xcentre ycentre xmajor ymajor xminor yminor*

Here *id* is the segment ID (an integer), while *xcentre, ycentre, xmajor, ymajor, xminor* and *yminor* are floating points.

".E" appends an ellipse to the specified segment.

(*xcentre*, *ycentre*) is the centre of the ellipse;
(*xmajor*, *ymajor*) is an extreme point along an axis of the ellipse;
(*xminor*, *yminor*) is an extreme point along the other axis.

.T *id x y text*

Here *id* is the segment ID (an integer), *x*, *y* are floating points and *text* is a quoted string.

".T" appends a text string to the specified segment.

(*x*, *y*) is the location of the start of the string.

.d *id*

.d *

Here *id* is the segment ID (an integer).

".d" deletes all attributes and entities having the segment ID *id*.

If * is used instead of *id*, ALL segments will be deleted.

.r

".r" clears the screen and repaints all segments.

.s *filename*

Here *filename* is a legal UNIX file name.

".s" saves all segments in a file (in text format).

5. The Scout System

The difference between the Scout system and the Scout-to-EDEN translator is that the Scout-to-EDEN translator only does the translation but the Scout system interprets inputs in Scout notation and produces displays on the screen. For the purpose of interpreting Scout inputs, the Scout system makes use of the Scout-to-EDEN translator, the EDEN interpreter and other tools.

The Scout system provides a coherent programming environment for definitive notations. Currently, the Scout system accepts input written in the definitive notations Scout and DoNaLD and the definitive language EDEN or the mixture of them. Because the Scout notation is intended to co-operate with other definitive notations, the Scout system is designed for easy extension. Therefore, it is important to understand the implementation of the current system as well as to know how to run the current system.

The Structure of the Scout System

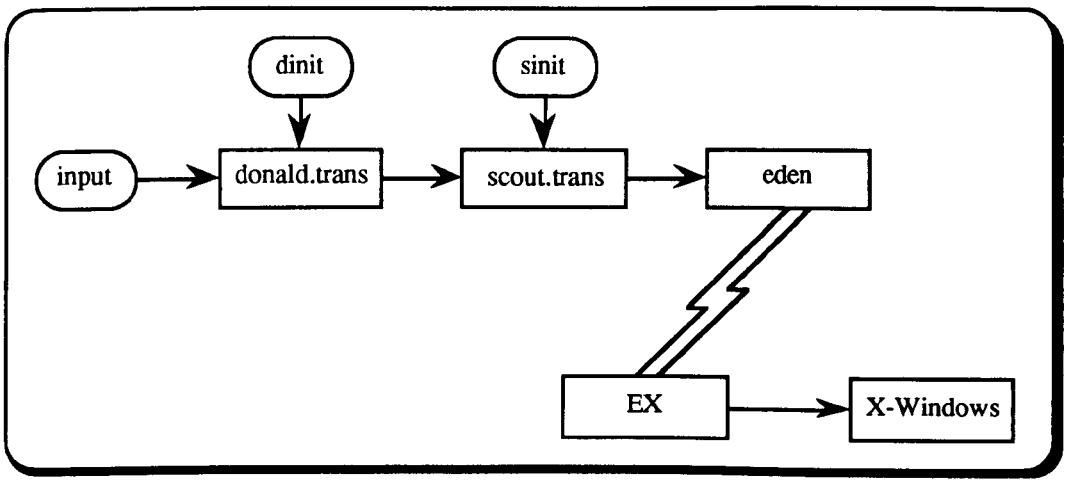


Figure 1: The Run-time Structure of the Scout System

To facilate simple extension of the system, a simple convention to the input script is derived – the piece of script written in a definitive notation must start with the line beginning with % and the notation name. Section 3 has already shown an illustrative

example. Under this convention, different translators only translate lines of their own notations into EDEN, leaving others untouched.

The advantage of having this convention is that the translators of different definitive notations may work independently of each other. Figure 1 is a set-up of the Scout system which is able to interpret input involving Scout, DoNaLD and EDEN. Because both `donald.trans` and `scout.trans` translate DoNaLD and Scout directly into EDEN, they will not interfere with each other, the position of `donald.trans` and `scout.trans` may be interchanged. In terms of UNIX commands:

cat script - | donald.trans dinit[4] | scout.trans sinit | eden -n[5]

and

cat script - | scout.trans sinit | donald.trans dinit | eden -n

have the same effect. Also if the input contains no DoNaLD or Scout notation, the corresponding translator may be omitted. Similarly, if a new translator is available (of course this translator must conform to the convention above), only an extra pipeline is required, no alteration to the existing usage of the system is necessary. (A translator from ARCA to EDEN can now be incorporated into the system in this scheme.)

Invocation of the Scout System

In order that the Scout system can be used conveniently, the Scout system makes use of shell scripts and symbolic links. These shell scripts and symbolic links are used to select the right files for different machine types, initialisation and window systems. In order to make the system relocatable, the shell scripts always reference the environment variable $PUBLIC. Therefore, you need to set the environment variable $PUBLIC to the directory containing the Scout system. At the time of writing this document, the path is /dcs/acad/wmb/public.

---

[4] "donald.trans dinit" is, in fact, syntactically wrong in our current system because donald.trans cannot take a file as an argument; it is so written here to conform to what scout.trans can do. Should type:

cat dinit - | donald.trans

to simulate the same effect.

[5] EX is a background process created through EDEN by a command in one of the initialisation files. Hence, EX does not appear in the UNIX command line. The -n option sets EDEN to no prompt mode.

To set $PUBLIC, if you are using csh, type:

```
setenv PUBLIC ~wmb/public
```

if you are using sh, type:

```
PUBLIC=/cs/acad/wmb/public
export PUBLIC
```

Files

$PUBLIC/	
bin/	
rq	message queue remover
eden	run this to start EDEN
scout	the Scout filter
donald	the DoNaLD filter
arca	the ARCA filter
scout.trans	alias to sun4/scout.trans
donald.trans	alias to sun4/donald.trans
arca.trans	alias to sun4/arca.trans
sun4/	sun4 version of sun3 executable files
EX	an EDEN/X Window interface program
c.eden	EDEN with curse library functions
scout.trans	Scout-to-EDEN translator
donald.trans	DoNaLD-to-EDEN translator
arca.trans	ARCA-to-EDEN translator
lib/	
scout/	
Eden-X/	
sinit	initialisation files for scout, will include scout.lib
scout.init	
donald/	
dinit	initialisation files for donald, will include xinit.e
xinit.e	
arca/	
ainit	initialisation files for arca, will include arca.lib
arca.lib	
ex/	
ex.init	file for starting up EX, will include ex.lib
ex.lib	
scout/	
Version2/	
Scout/	source directory for the Scout translator
EX/	source directory for EX
demo/	demonstration programs

Files scout.log, s.output, d.output and a.output will be created in the working directory. They are the log files for the Scout input, the output of the Scout-to-

EDEN translator, the output of the DoNaLD-to-EDEN translator and the output of the ARCA-to-EDEN translator respectively.

Example

Supposing that there is a demonstration file *demo* to be executed and that $PUBLIC/bin is included in the path.

If demo contains both DoNaLD and Scout definitions, do

```
cat demo - | donald | scout | eden -n
```

or

```
cat demo - | scout | donald | eden -n
```

if demo contains only Scout definitions (with/without EDEN definitions), do

```
cat demo - | scout | eden -n
```

if demo contains only DoNaLD definitions (with/without EDEN definitions), do

```
cat demo - | donald | eden -n
```

Notes

- The system is now running under X11R5.

Bugs

- Occasionally, the message queue in use may be still active even when the system has terminated. A user should check using the UNIX command `ipcs` to check if this is the case. If so, the command `rq` may be executed to remove the message queues (usage: rq *start end* (remove message queues numbered *start* through *end*).

- Because EX and the rest of the Scout system are loosely linked through message queue, EX may be still running when the rest of the system is abnormally terminated. For this reason, there is an `EX-Killer` window created when EX is initiated. Pressing the button in the window will terminate EX.

- Message queue is an IPC option of the SUNOS, therefore, the Scout system may not be portable to other sites.

- The size of the message queue is set to 1K (in the source program of EX). Should a text window in the Scout definitions consist of a string of about 700 characters, error may occur.

6. Changes to DoNaLD and EDEN

Changes to DoNaLD

- There is a new declaration in DoNaLD, called `viewport`.

  ```
  viewport vp_name
  ```

  declares that the following DoNaLD definitions are part of the picture named *vp_name*. To display this DoNaLD drawing, the `pict` field of the window displaying this drawing should be defined as: `pict: "vp_name"`.

  If no viewport is defined, the drawing will be displayed on a separated X-window independent of Scout.

- The maximum number of openshapes is 128.

- The meaning of the `scale` function has been changed. `Scale` now scales all the items of an object relative to the origin in DoNaLD's coordinate system rather than to the centre of the object (e.g. mid-point of a line).

- A new translation function is added. Usage: `trans(object, x, y)`

- A new data type – `ellipse` – is added. Usage:

  ```
  ellipse e
  e = ellipse(p0, p1, p2)
  ```

  where p0 = the centre of the ellipse
  p1 = an extreme point along an axis of the ellipse
  p2 = an extreme point along the other axis of the ellipse

Changes to EDEN

- All lines beginning with % are regarded as comments.

- New functions:

  time() return the current time in seconds since Jan 1, 1970

ftime() return a list of two integers, [*seconds*, *milli*] where
seconds is the tme in seconds and
milli is the number of milli-seconds in addition to the time elapsed since Jan 1, 1970

gettime() return the current time as a list of seven integers. The meaning of these integers and their ranges are, in their order:

second	0–59
minute	0–59
hour	0 – 23
day of month	1 – 31
month of year	1 – 12
year	year – 1900
day of week	0–6 (0 = Sunday)

- Other changes like the functions for message queue operation can be found in $PUBLIC/EDEN/VERSION

References

[1] Beynon, W M. "Definitive Notations for Interaction", Proc. BCS Conference "People and Computers: Designing the Interface", ed. Johnson and Cook, CUP, 1985

[2] Beynon, W M. "Definitive Principles for Interactive Graphics", NATO ASI Series F: 140, 1988

[3] Beynon, W M. "Evaluating Definitive Principles for Interaction in Graphics", University of Warwick Computer Science Research Report 133, November 1988

[4] Yung, Y P. "Software Experiments Using the Definition-Based State-Transition Model", MSc thesis, Computer Science Department, University of Warwick, 1991

[5] Yung, Y W. "EDEN: An Engine for Definitive Notations. Design, Implementation and Evaluation", MSc. thesis, Department of Computer Science, University of Warwick, 1989

Appendix B

User Guide to Admira

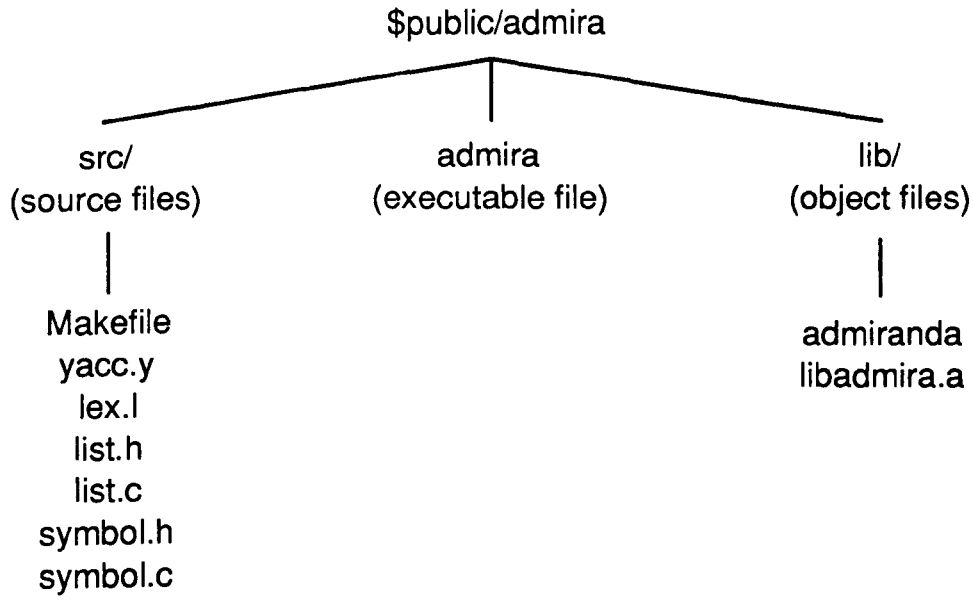
User Guide to Admira

1. Notes on Access to *Admira*

To run *admira*:

You may run the *admira* system by executing the file $public/admira at any directory where $public represents the directory containing all the definitive projects, currently /dcs/acad/wmb/public.

The Architecture of the Admira System



Maintenance of the *admira* system:

You may be interested in making your own copy of the *admira* system. You are suggested to keep the above file architecture. What you have to do is to:

1. copy the whole directory $public/admira to your area.
2. change the value of the variable ADMIRA in the file admira to the directory containing admira itself.
3. change the value of the variable ROOT in the file Makefile to the directory containing admira as well.

2. Syntax of *Admira* Dialogue and Expressions

dialogue::= defs .
synonym .
spec .
query .

synonym::= tform == type

query::= ? exp

def::= fnform = rhs

defs::= def
defs ; def

rhs::= exp whdefs?
cases whdefs?

cases::= alt ; = cases
lastcase

alt::= exp , exp

lastcase::= alt
exp , otherwise

whdefs::= where defs end { In *Miranda*, it is: where defs }

spec::= var :: type

type::= argtype
typename argtype*
type -> type

argtype::= typename
typevar
(type_list?)
[type_list]

tform::= typename typevar*

```
fnform::= var formal*

pat::=  formal
        pat : pat
        pat + numeral

formal::= var
          literal
          ( pat_list? )
          [ pat_list? ]

exp::= e1
       prefix
       infix

e1::= simple+
      prefix e1
      exp infix e1

simple::= var
          literal
          show
          ( infix exp )
          ( exp infix )
          ( exp_list? )
          [ exp_list? ]
          [ exp .. exp? ]
          [ exp , exp .. exp? ]
          [ exp | qualifiers ]
          [ exp // qualifiers ]

qualifiers::= qualifier ; qualifiers
              qualifier

qualifier::= exp
             generator

generator::= pat_list <- exp          { In yacc, use: exp_list <- exp          }
             pat <- exp , exp ..      { In yacc, use: exp_list <- exp , exp .. }

var::= identifier
```

typename::= identifier

literal::= numeral
charconst
stringconst

Comments on the syntax for *admira* scripts

The syntax of *admira* is a subset of *Miranda* with some minor modifications. Changes include:

(1) An end is needed in the where clause.

(2) No user defined types is allowed.

(3) No abstract data types is allowed.

(4) If more than one variable is going to be declared for the same type, they must be declared separately.

(5) The use of colons in *Miranda* is optional while the use of colons in *admira* must be carefully observed.

(6) Formulae for describing a definition (those within where clauses and those using pattern matchings) have to be separated by colons.

(7) There is query statement in *admira* which is not presented in *Miranda* script.

(8) Definitions and queries are terminated by '.'.

The original production rules of Miranda will lead to some reduce/reduce conflicts in yacc. To simplify the implementation, the comments in the braces are the suggested productions for use in yacc.

Key to abbreviations in syntax:

decl = declaration,	tform = typeform,	def = definition,
spec = specification,	fnform = function form,	rhs = right hand side,
pat = pattern,	var = variable,	whdefs = where defs,
alt = alternative,	exp = expression,	e1 = operator expression

Conventions

We use a variant of BNF, in which non-terminals are represented by lower case words, and alternative productions are written on successive lines. (These departures from convention are adopted because '|' is concrete symbol of the language.)

For any non-terminal x,

x^*	means zero or more occurrences of x
x^+	means one or more occurrences of x
$x^?$	means the presence of x is optional
x-list	means one or more x's (separated by commas if >1)

A 'typevar' is a sequence of one or more stars (eg '*', '**' etc).

Operators

Here is a list of all prefix and infix operators, in order of increasing binding power. Operators given on the same line are of the same binding power. Prefix operators are identified as such in the comments - all others are infix.

operator	*comments*
: ++ --	right associative
\/	associative
&	associative
~	prefix
> >= = ~= <= <	continued relations allowed, eg 0<x<=10
+ -	left associative
-	prefix
* / div mod	left associative
^	right associative
.	associative
#	prefix
!	left associative

Brief explanation of each operator:

: prefix an element to a list, type *->[*]->[*]

++ -- list concatenation, list subtraction, both of type [*]->[*]->[*]

\/ & logical 'or', 'and', both of type bool->bool->bool

~ logical negation, type bool->bool

> >= = ~= <= <

comparison operators, all of type *->*->bool

Note that there is an ordering defined on every (non-function) type. In the case of numbers, characters and strings the order is as you would expect, on other types it as an arbitrary but reproducible ordering. Equality on structured data is a test for isomorphism. (i.e. in LISP terms it is "EQUAL" not "EQ"). It is an error to test functions for equality or order.

+ - plus, minus, type num->num->num

- unary munus, type num->num

Note that in *Miranda* unary minus binds less tightly than the multiplication and division operators. This is the usual algebraic convention, but is different from PASCAL.

* / div mod

times, divide, integer divide, integer remainder, all of type num->num->num

^ 'to the power of', type num->num->num

. function composition, type (**->***)->(*->**)->*->***

length of list, type [*]->num

! list subscripting, type [*]->num->*

note that the first element of a non-empty list x is x!0 and the last element is

x!(#x-1)

Appendix C

Program Listing of Admira

```
1   /*************************************************************************
2    * FILE:        yacc.y                                                   *
3    * DESCRIPTION: parser generator for admira and main loop                *
4    * AUTHOR:      Simon Y P Yung                                           *
5    *************************************************************************/
6
7   %{
8   #include <setjmp.h>
9   #include <stdio.h>
10  #include "list.h"
11  #include "symbol.h"
12
13  jmp_buf start;
14  extern char *head_of_instring();
15  %}
16  %union {
17          char    *name;
18          vlist   *l;
19          vlist2   ll;
20  }
21
22  %token DIV MOD OTHERWISE SHOW WHERE END
23  %token NUMERAL CHAR STRING SYNONYM SPEC DIAGONAL LARROW
24  %token DOTDOT CONCAT SUBTRACT OR GE LE NE TYPEVAR TERMINAL
25  %token <name> ID
26
27  %type <l> query rhs cases alt lastcase type type_list
28  %type <l> argtype argtype_star tform pat pat_list formal formal_star
29  %type <l> exp exp_list e1 simple simple_plus var typename
30  %type <ll> synonym def def_plus whdefs spec fnform
31  %type <ll> qualifiers qualifier generator
32
33  %right RARROW
34  %right ':' CONCAT SUBTRACT
35  %left  OR
36  %left  '&'
37  %left  '~'
38  %left  '>' GE '=' NE LE '<'
39  %left  '+' '-'
40  %left  UMINUS
41  %left  '*' '/' DIV MOD
42  %right '^'
43  %left  '.'
44  %left  '#'
45  %left  '!'
46
47  %%
48  dialogue :
49            def_plus TERMINAL      { define($1);   return 1; }
50          | synonym  TERMINAL      { define($1);   return 1; }
51          | spec     TERMINAL      { declare($1);  return 1; }
52          | query    TERMINAL      { query($1);    return 1; }
53          | error    TERMINAL      {
54                                     yyerrok;
55                                     return 1;
56                                   }
57          ;
58
59  synonym :
60            tform SYNONYM type     { $$.b = $1; $$.v = $3; }
61          ;
62
63  query :
64            '?' exp                { $$ = $2; }
65          ;
66
67  def :
68            fnform '=' rhs         { $$.b = $1.b;
69                                     $$.v = listsub(listsub($3, $1.v), $1.b);
70                                     freevlist($1.v); }
71          ;
```



```
72
73  def_plus :
74            def                    { $$ = $1; }
75          | def_plus ';' def       { $$.b = listadd($1.b, $3.b);
76                                     $$.v = listadd($1.v, $3.v); }
77          ;
78
79  rhs :
80            exp whdefs             { $$ = listsub(listadd($1, $2.v), $2.b);
81                                     freevlist($2.b); }
82          | exp                    { $$ = $1; }
83          | cases whdefs           { $$ = listsub(listadd($1, $2.v), $2.b);
84                                     freevlist($2.b); }
85          | cases                  { $$ = $1; }
86          ;
87
88  cases :
89            alt ';' '=' cases      { $$ = listadd($1, $4); }
90          | lastcase               { $$ = $1; }
91          ;
92
93  alt :
94            exp ',' exp            { $$ = listadd($1, $3); }
95          ;
96
97  lastcase :
98            alt                    { $$ = $1; }
99          | exp ',' OTHERWISE      { $$ = $1; }
100         ;
101
102 whdefs :
103           WHERE def_plus END     { $$ = $2; }
104         ;
105
106 spec :
107           var SPEC type          { $$.b = $1; $$.v = $3; }
108         ;
109
110 type :
111           typevar                { $$ = emptylist(); }
112         | '(' type_list ')'      { $$ = $2; }
113         | '(' ')'                { $$ = emptylist(); }
114         | '[' type_list ']'      { $$ = $2; }
115         | typename argtype_star  { $$ = listadd($1, $2); }
116         | type RARROW type       { $$ = listadd($1, $3); }
117         ;
118
119 type_list :
120           type                   { $$ = $1; }
121         | type_list ',' type     { $$ = listadd($1, $3); }
122         ;
123
124 argtype :
125           typename               { $$ = $1; }
126         | typevar                { $$ = emptylist(); }
127         | '(' type_list ')'      { $$ = $2; }
128         | '(' ')'                { $$ = emptylist(); }
129         | '[' type_list ']'      { $$ = $2; }
130         ;
131
132 argtype_star :                   { $$ = emptylist(); }
133         | argtype_star argtype   { $$ = listadd($1, $2); }
134         ;
135
136 tform :
137           typename typevar_star  { $$ = $1; }
138         ;
139
140 fnform :
141           var formal_star        { $$.b = $1; $$.v = $2; }
142         ;
```

```
143
144   pat :
145             formal                  { $$ = $1; }
146           | pat ':' pat             { $$ = listadd($1, $3); }
147           | pat '+' NUMERAL         { $$ = $1; }
148           ;
149
150   pat_list :
151             pat                     { $$ = $1; }
152           | pat_list ',' pat        { $$ = listadd($1, $3); }
153           ;
154
155   formal :
156             var                     { $$ = $1; }
157           | literal                 { $$ = emptylist(); }
158           | '(' pat_list ')'        { $$ = $2; }
159           | '(' ')'                 { $$ = emptylist(); }
160           | '[' pat_list ']'        { $$ = $2; }
161           | '[' ']'                 { $$ = emptylist(); }
162           ;
163
164   formal_star :                     { $$ = emptylist(); }
165           | formal_star formal      { $$ = listadd($1, $2); }
166           ;
167
168   exp :
169             e1                      { $$ = $1; }
170           | '~'                     { $$ = emptylist(); }
171           | '#'                     { $$ = emptylist(); }
172           | infix                   { $$ = emptylist(); }
173           ;
174
175   e1 :
176             simple_plus             { $$ = $1; }
177           | '~' e1                  { $$ = $2; }
178           | '~' e1 %prec UMINUS     { $$ = $2; }
179           | '#' e1                  { $$ = $2; }
180           | e1 infix e1             { $$ = listadd($1, $3); }
181           ;
182
183   exp_list :
184             exp                     { $$ = $1; }
185           | exp ',' exp_list        { $$ = listadd($1, $3); }
186           ;
187
188   infix :
189             ':'   | CONCAT        | SUBTRACT      | OR
190           | '&'   | '>'   | GE    | '='   | NE    | LE
191           | '<'   | '+'   | '-'   | '*'   | '/'   | DIV
192           | MOD   | '^'   | '.'   | '!'
193           ;
194
195   simple :
196             var                     { $$ = $1; }
197           | literal                 { $$ = emptylist(); }
198           | SHOW                    { $$ = emptylist(); }
199           | '(' infix exp ')'       { $$ = $3; }
200           | '(' exp infix ')'       { $$ = $2; }
201           | '(' exp_list ')'        { $$ = $2; }
202           | '(' ')'                 { $$ = emptylist(); }
203           | '[' exp_list ']'        { $$ = $2; }
204           | '[' ']'                 { $$ = emptylist(); }
205           | '[' exp DOTDOT ']'      { $$ = $2; }
206           | '[' exp DOTDOT exp ']'
207                 { $$ = listadd($2, $4); }
208           | '[' exp ',' exp DOTDOT exp ']'
209                 { $$ = listadd(listadd($2, $4), $6); }
210           | '[' exp ',' exp DOTDOT ']'
211                 { $$ = listadd($2, $4); }
212           | '[' exp '|' qualifiers ']'
213                 { $$ = listsub(listadd($2, $4.v), $4.b);
```



```
214                   freevlist($4.b); }
215           | '[' exp DIAGONAL qualifiers ']'
216                 { $$ = listsub(listadd($2, $4.v), $4.b);
217                   freevlist($4.b); }
218           ;
219
220   simple_plus :
221             simple                  { $$ = $1; }
222           | simple_plus simple      { $$ = listadd($1, $2); }
223           ;
224
225   qualifiers :
226             qualifier               { $$ = $1; }
227           | qualifiers ';' qualifier
228                                     { $$.b = listadd($1.b, $3.b);
229                                       $$.v = listadd($1.v, $3.v); }
230           ;
231
232   qualifier :
233             exp                     { $$.b = emptylist();
234                                       $$.v = $1;            }
235           | generator               { $$ = $1; }
236           ;
237
238   generator :
239             exp_list LARROW exp
240                   { $$.b = $1; $$.v = $3; }
241           | exp_list LARROW exp ',' exp DOTDOT
242                   { $$.b = $1; $$.v = listadd($3, $5); }
243           ;
244
245   var :
246             ID                      { $$ = mklist($1); }
247           ;
248
249   typename :
250             ID                      { $$ = mklist($1); }
251           ;
252
253   typevar :
254             '*'
255           | TYPEVAR
256           ;
257
258   typevar_star :
259           | typevar_star typevar
260           ;
261
262   literal :
263             NUMERAL
264           | CHAR
265           | STRING
266           ;
267   %%
268   errmsg(s)
269       char   *s;
270   {
271       extern char *instring;
272       fprintf(stderr, "Error: %s\n", s);
273       free(instring);
274       instring = "";
275       return;
276   }
277
278   yyerror()
279   {
280
281       if (feof(stdin)) {
282           fprintf(stderr, "Bye\n");
283           exit(0);
284       } else {
```

```
285             errmsg("parse error");
286             return;
287         }
288     }
289
290     main()
291     {
292         setbuf(stdout, NULL);
293         setjmp(start);
294         do {
295             fprintf(stderr, "-> ");
296         } while (yyparse());
297         fprintf(stderr, "BYE\n");
298     }
```

```
%{
/*****************************************************************************
 * FILE:        lex.l                                                        *
 * DESCRIPTION: lexical analyzer generator for admira                        *
 * AUTHOR:      Simon Y P Yung                                               *
 *****************************************************************************/

#include <stdio.h>
#include <strings.h>
#include "list.h"
#include "y.tab.h"

#undef ECHO
#define ECHO instring = stringcat(instring, yytext)
char *instring = "";

char   *
strsave(s)
    char   *s;
{
    char   *ptr;

    ptr = (char *) malloc(strlen(s) + 1);
    return strcpy(ptr, s);
}

char   *
head_of_instring()
{
    char   *head;

    head = instring;
    instring = index(instring, '@');
    *instring++ = '\0';
    return head;
}

char   *
stringcat(s1, s2)
char   *s1, *s2;
{
    char *s;

    s = (char *)malloc(strlen(s1) + strlen(s2) + 1);
    strcpy(s, s1);
    strcat(s, s2);
    free(s1);
    return s;
}
%}

D                       [0-9]
E                       e-?(D)+
LETTER                  [A-Za-z]
ALPHANUM                [0-9A-Za-z_']
SPACE                   [ \t]

%%
"."[ \t]*\n             { instring = stringcat(instring, "\n@");
                          return TERMINAL;      }
div                     { ECHO; return DIV; }
mod                     { ECHO; return MOD; }
otherwise               { ECHO; return OTHERWISE; }
show                    { ECHO; return SHOW; }
where                   { ECHO; return WHERE; }
end                     { return END; }
\"[^"]*                 {
                          if (yytext[yyleng-1] == '\\') {
                               yymore();
                          } else {
                               input();
                               yytext[yyleng++] = '"';
                               yytext[yyleng++] = '\0';
                               ECHO;
                               return STRING;
                          }
                        }
\'[^']*                 {
                          if (yytext[yyleng-1] == '\\') {
                               yymore();
                          } else {
                               input();
                               yytext[yyleng++] = '\'';
                               yytext[yyleng++] = '\0';
                               ECHO;
                               return CHAR;
                          }
                        }
-?(D)+((E))?            |
-?(D)*"."(D)+((E))?     { ECHO; return NUMERAL; }
(LETTER)(ALPHANUM)*     {
                          ECHO;
                          yylval.name = strsave(yytext);
                          return ID;
                        }
"=="                    { ECHO; return SYNONYM; }
"::"                    { ECHO; return SPEC;    }
"//"                    { ECHO; return DIAGONAL;}
"<-"                    { ECHO; return LARROW;  }
"->"                    { ECHO; return RARROW;  }
".."                    { ECHO; return DOTDOT;  }
"++"                    { ECHO; return CONCAT;  }
"--"                    { ECHO; return SUBTRACT;}
"\\/"                   { ECHO; return OR;      }
">="                    { ECHO; return GE;      }
"<="                    { ECHO; return LE;      }
"~="                    { ECHO; return NE;      }
"*****"+                { ECHO; return TYPEVAR; }
(SPACE)+                { ECHO; }
\n                      { ECHO; fprintf(stderr, " > "); }
.                       { ECHO; return yytext[0];       }
%%
```

```
1   /*****************************************************************************
2    * FILE:         list.c                                                      *
3    * DESCRIPTION: list construction and destruction functions                  *
4    * AUTHOR:       Simon Y P Yung                                              *
5    *****************************************************************************/
6
7   #include "list.h"
8
9   /*****************************************************************************
10      emptylist - create a null list
11   *****************************************************************************/
12  vlist  *
13  emptylist()
14  {
15      return NIL;
16  }
17
18  /*****************************************************************************
19      mklist - create a single element list storing 'name'
20   *****************************************************************************/
21  vlist  *
22  mklist(name)
23      char   *name;
24  {
25      vlist  *ptr;
26
27      ptr = (vlist *) malloc(sizeof(vlist));
28      ptr->name = name;
29      ptr->next = NIL;
30      return ptr;
31  }
32
33  /*****************************************************************************
34      listadd - append l2 to l1
35   *****************************************************************************/
36  vlist  *
37  listadd(l1, l2)
38      vlist  *l1, *l2;
39  {
40      vlist  *ptr;
41
42      if (l1 == NIL)
43          return l2;
44      if (l2 == NIL)
45          return l1;
46      for (ptr = l1; ptr->next != NIL; ptr = ptr->next);
47      ptr->next = l2;
48      return l1;
49  }
50
51  /*****************************************************************************
52      listsub - remove from l1 those symbols in l2
53   *****************************************************************************/
54  vlist  *
55  listsub(l1, l2)
56      vlist  *l1, *l2;
57  {
58      vlist  *ptr;
59      vlist  *l1_ptr;
60      vlist  *l2_ptr;
61
62      for (l2_ptr = l2; l2_ptr != NIL; l2_ptr = l2_ptr->next) {
63          while (l1 != NIL && strcmp(l1->name, l2_ptr->name) == 0) {
64              ptr = l1->next;
65              free(l1->name);
66              free(l1);
67              l1 = ptr;
68          }
69          if (l1 == NIL)
70              break;
71          l1_ptr = l1;
```



```
72          while (l1_ptr->next != NIL) {
73              ptr = l1_ptr->next;
74              if (strcmp(ptr->name, l2_ptr->name) == 0) {
75                  free(ptr->name);
76                  l1_ptr->next = ptr->next;
77                  free(ptr);
78              } else {
79                  l1_ptr = ptr;
80              }
81          }
82      }
83      return l1;
84  }
85
86  /*****************************************************************************
87      freevlist - free the whole list and
88                  the strings pointed by its elements
89   *****************************************************************************/
90  freevlist(l)
91      vlist  *l;
92  {
93      vlist  *ptr;
94      vlist  *l_ptr;
95
96      for (l_ptr = l; l_ptr != NIL; l_ptr = ptr) {
97          ptr = l_ptr->next;
98          free(l_ptr->name);
99          free(l_ptr);
100     }
101     return;
102 }
```

```
1   /*************************************************************************
2    * FILE:         symbol.c                                                *
3    * DESCRIPTION:  Implementation of symbol table and the functions that   *
4    *               access it                                               *
5    * AUTHOR:       Simon Y P Yung                                          *
6    *************************************************************************/
7
8   #include <stdio.h>
9   #include <strings.h>
10  #include <signal.h>
11  #include <setjmp.h>
12  #include "list.h"
13
14  jmp_buf wait;
15
16  #define MAXVAR 256                      /* max size of symbol table      */
17
18  typedef struct symbol {
19      char            *name;              /* name of the symbol            */
20      char            *def;               /* definition of the symbol      */
21      char            *type;              /* type declaration of the symbol */
22      int              dep[MAXVAR];       /* dependency array of the definition */
23      int              tdep[MAXVAR];      /* type dependency array         */
24      int              inuse;             /* use only at run-time          */
25  } symbol;
26
27  static symbol   table[MAXVAR];
28  static int      end_of_table = 0;       /* end of symbol table           */
29  static FILE     *fp;                    /* miranda script                */
30
31  extern jmp_buf  start;
32  extern char     *head_of_instring();
33  extern char     *strsave();
34
35  void cont() {
36          (void) signal(SIGINT, cont);
37          longjmp(wait, 1);
38  }
39
40  /*************************************************************************
41      lookup - return the entry in the symbol table
42      name - name of symbol
43   *************************************************************************/
44  int
45  lookup(name)
46      char            *name;
47  {
48      int              i;
49
50      for (i = 0; i < end_of_table; i++) {
51          if (strcmp(name, table[i].name) == 0) {
52              return i;                   /* name already in the table */
53          }
54      }
55      if (end_of_table == MAXVAR) {
56          errmsg("out of memory");
57          longjmp(start);
58      }
59      /* create a new entry */
60      table[end_of_table].name = strsave(name);
61      table[end_of_table].def = NULL;
62      table[end_of_table].type = NULL;
63      for (i = 0; i < MAXVAR; i++) {
64          table[end_of_table].dep[i] = 0;
65          table[end_of_table].tdep[i] = 0;
66      }
67      table[end_of_table].inuse = 0;
68      return end_of_table++;
69  }
70
71  /*************************************************************************
```



```
72      define - (re)define the symbol in l.b
73      l.b - the list containing the name of the symbol to be (re)defined
74            if l.b has more than one symbol(in case of pattern matching),
75            all names should be the same
76      l.v - list of the names of the unbounded symbols in the definition
77   *************************************************************************/
78  define(l)
79      vlist2          l;
80  {
81      int             i, j;
82      vlist           *ptr;
83
84      /* check for multiple definition */
85      for (ptr = l.b->next; ptr != NIL; ptr = ptr->next) {
86          if (strcmp(l.b->name, ptr->name)) {
87              errmsg("name inconsistent in pattern matching");
88              freevlist(l.b);
89              freevlist(l.v);
90              return;
91          }
92      }
93
94      i = lookup(l.b->name);
95      if (table[i].def != NULL)
96          free(table[i].def);             /* clear the old definition      */
97      for (j = 0; j < MAXVAR; j++)
98          table[i].dep[j] = 0;            /* clear the dependency array    */
99      table[i].def = head_of_instring();  /* put in new definition         */
100     for (ptr = l.v; ptr != NIL; ptr = ptr->next) {
101         table[i].dep[lookup(ptr->name)] = 1;    /* maintain new dependency */
102     }
103     freevlist(l.b);
104     freevlist(l.v);
105     return;
106 }
107
108 /*************************************************************************
109     declare - (re)declare the type of the symbol in l.b
110     l.b - a single element list containing the name of the symbol
111           to be (re)declared
112     l.v - list of the names of the unbounded symbols in the declaration
113  *************************************************************************/
114 declare(l)
115     vlist2          l;
116 {
117     int             i, j;
118     vlist           *ptr;
119
120     i = lookup(l.b->name);
121     if (table[i].type != NULL)
122         free(table[i].type);            /* clear the old declaration     */
123     for (j = 0; j < MAXVAR; j++)
124         table[i].tdep[j] = 0;           /* clear the dependency array    */
125     table[i].type = head_of_instring(); /* put in new declaration        */
126     for (ptr = l.v; ptr != NIL; ptr = ptr->next) {
127         table[i].tdep[lookup(ptr->name)] = 1;   /* maintain new dependency */
128     }
129     freevlist(l.b);
130     freevlist(l.v);
131     return;
132 }
133
134 /*************************************************************************
135     query - evaluate an expression
136     l - names of the symbols unbounded in the expression
137  *************************************************************************/
138 query(l)
139     vlist           *l;
140 {
141     char            *exp;
142     int              i;
```

```
143     vlist           *ptr;
144     char             script[18];
145     FILE            *feedback;
146     char             dummy[2];
147
148     strcpy(script, "/tmp/miraXXXXXX");
149     strcat(mktemp(script), ".m");       /* create name of miranda script*/
150     fp = fopen(script, "w");
151     if (fp == NULL) {
152         perror(script);
153     } else {
154         if (signal(SIGINT, SIG_IGN) != SIG_IGN)
155                 (void)signal(SIGINT, cont);
156
157         /* generate miranda script */
158         for (i = 0; i < end_of_table; i++)
159             table[i].inuse = 0;
160         for (ptr = l; ptr != NIL; ptr = ptr->next)
161             printdef(lookup(ptr->name));
162         fclose(fp);
163
164         printf("/file %s\n", script);   /* ask mira to read the script  */
165
166         exp = head_of_instring();
167         printf("%s", index(exp, '?') + 1);      /* ask mira to evaluate exp */
168         printf("!kill -INT %d\n", getpid());
169
170         if (!setjmp(wait)) {
171                 for (;;);       /* wait for mira to signal end of eval */
172         }
173
174         /* remove the temporary files */
175         /*
176         unlink(script);
177         script[16] = 'x';
178         unlink(script);
179         */
180
181         free(exp);
182     }
183     freevlist(l);
184 }
185
186 /************************************************************************
187     printdef - recursively write the definitions and/or declarations of
188                the symbol i and those it depends
189     i - symbol table entry
190  ************************************************************************/
191 printdef(i)
192     int             i;
193 {
194     int             j;
195
196     if (table[i].inuse)
197         return;
198     table[i].inuse = 1;
199     for (j = 0; j < end_of_table; j++)
200         if (table[i].dep[j] || table[i].tdep[j])
201             printdef(j);
202     if (table[i].type != NULL)
203         fprintf(fp, "%s", table[i].type);
204     if (table[i].def != NULL)
205         fprintf(fp, "%s", table[i].def);
206     return;
207 }
```

```
1   /*****************************************************************************
2    * FILE:        list.h                                                       *
3    * DESCRIPTION: header file of list.c                                        *
4    * AUTHOR:      Simon Y P Yung                                               *
5    *****************************************************************************/
6
7   typedef struct vlist {
8       char   *name;
9       struct vlist *next;
10  }       vlist;                  /* list of variables                    */
11
12  typedef struct vlist2 {
13      vlist  *b;
14      vlist  *v;
15  }       vlist2;                 /* collection of 2 lists of variables   */
16
17  extern vlist *listadd();        /* concat 2nd vlist to 1st vlist        */
18  extern vlist *listsub();        /* remove variables in 2nd vlist from
19                                     the 1st vlist                        */
20  extern vlist *emptylist();      /* create an empty vlist                */
21  extern vlist *mklist();         /* create a vlist of 1 variable         */
22  extern  freevlist();            /* free the memory occupied by vlist    */
23
24  #define NIL (vlist *)0
```

```
1     /*************************************************************************
2      * FILE:        symbol.h                                                 *
3      * DESCRIPTION: routines for manipulating the symbol table               *
4      * AUTHOR:      Simon Y P Yung                                           *
5      *************************************************************************/
6
7     extern          define();       /* put a definition into symbol table   */
8     extern          declare();      /* put a declaration into symbol table  */
9     extern          query();        /* evaluate an expression               */
```

Appendix D

The Jugs Example

D.1. EDEN Definitions and Actions

D.2. Scout Display Specification

D.3. Original Display Specification

The Jugs Example

D.1. EDEN Definitions and Actions

```
1     func max ( return $1<$2 ? $2 : $1; );
2
3     target=1;
4     capA = 5;
5     capB = 7;
6     Afull is capA==contentA;
7     Bfull is capB==contentB;
8     contentA = 0;
9     contentB = 0;
10
11    height is max(capA,capB)+2;
12
13    widthB = 5;
14    widthA = 5;
15    menu is ["1:Fill A","2:Fill B","3:Empty A","4:Empty B","5:Pour"];
16    menustatus is [valid1, valid2, valid3, valid4, valid5, valid6, valid7];
17                                  /* two invisible options 6 & 7 */
18    valid1 is !Afull;
19    valid2 is !Bfull;
20    valid3 is contentA != 0;
21    valid4 is contentB != 0;
22    valid5 is valid6 || valid7;
23    valid6 is valid3 && valid2;
24    valid7 is valid4 && valid1;
25
26    /* specifying the control information */
27
28    Error=0; updating=0;
29    finish is ((contentA==target)||(contentB==target))&&!updating;
30
31    func avail
32    {
33            /* indicates whether the menu option with parameter $1 is open */
34            auto t;
35            t = menustatus[$1];
36            return t;
37    }
38
39
40    proc init_pour : input
41    {
42            updating = 1;
43            if (input == 5) {
44                    content5 = contentA + contentB;
45                    contentB is content5 - contentA;
46                    option = valid6 ? 6 : 7;
47            } else
48                    option = input;
49            step = 0;
50    }
51
52    proc pour :  step
53    {
54            if (avail(option)) {
55                    switch (option) {
56                    case 1:
57                            contentA = contentA + 1;
58                            break;
59                    case 2:
60                            contentB = contentB + 1;
61                            break;
62                    case 3:
63                            contentA = contentA - 1;
64                            break;
65                    case 4:
66                            contentB = contentB - 1;
67                            break;
68                    case 6:
69                            contentA = contentA - 1;
70                            break;
71                    case 7:
```



```
72                          contentA = contentA + 1;
73                          break;
74                  default:
75                          writeln("option = ", option);
76                          return;
77                  }
78                  eager();
79                  step++;
80          } else {
81                  contentA = contentA;
82                  contentB = contentB;
83                  updating = 0;
84          }
85    }
86
87    status is (Error)?"invalid option": ((updating)?"updating":"awaiting input");
88    totstat is (finish) ? "Success!" : status;
89    targ is ("Target is " // str(target) // " : ");
90    stat is messdisplay(height,totstat);
91    targt is messdisplay(height,targ);
```

The Jugs Example

D.2. Scout Display Specification

```
%eden
include("SCOUT/tank.e");
%scout
string menu1, menu2, menu3, menu4, menu5;
%eden
menu1 is menu[1];
menu2 is menu[2];
menu3 is menu[3];
menu4 is menu[4];
menu5 is menu[5];
%scout

string valid_fg, valid_bg, invalid_fg, invalid_bg;
valid_fg = "black";
valid_bg = "white";
invalid_fg = "white";
invalid_bg = "black";

integer valid1, valid2, valid3, valid4, valid5;

window wmenu1, wmenu2, wmenu3, wmenu4, wmenu5;
box bmenu1, bmenu2, bmenu3, bmenu4, bmenu5;
point base;

base = (1.c, 20.r);

wmenu1 = {
        frame:  (bmenu1),
        string: menu1,
        fgcolor:if valid1 then valid_fg else invalid_fg endif,
        bgcolor:if valid1 then valid_bg else invalid_bg endif
};
bmenu1 = [base, 1, strlen(menu1)];

wmenu2 = {
        frame:  (bmenu2),
        string: menu2,
        fgcolor:if valid2 then valid_fg else invalid_fg endif,
        bgcolor:if valid2 then valid_bg else invalid_bg endif
};
bmenu2 = [bmenu1.ne + (1.c, 0), 1, strlen(menu2)];
# bmenu2 is one space right of bmenu1

wmenu3 = {
        frame:  (bmenu3),
        string: menu3,
        fgcolor:if valid3 then valid_fg else invalid_fg endif,
        bgcolor:if valid3 then valid_bg else invalid_bg endif
};
bmenu3 = [bmenu2.ne + (1.c, 0), 1, strlen(menu3)];
# bmenu3 is one space right of bmenu2

wmenu4 = {
        frame:  (bmenu4),
        string: menu4,
        fgcolor:if valid4 then valid_fg else invalid_fg endif,
        bgcolor:if valid4 then valid_bg else invalid_bg endif
};
bmenu4 = [bmenu3.ne + (1.c, 0), 1, strlen(menu4)];
# bmenu4 is one space right of bmenu3

wmenu5 = {
        frame:  (bmenu5),
        string: menu5,
        fgcolor:if valid5 then valid_fg else invalid_fg endif,
        bgcolor:if valid5 then valid_bg else invalid_bg endif
};
bmenu5 = [bmenu4.ne + (1.c, 0), 1, strlen(menu5)];
# bmenu5 is one space right of bmenu4

integer capA, capB, contentA, contentB, widthA, widthB, height;
```



```
string targ, totstat;

window wcapA, wcapB, wcontentA, wcontentB;
frame fcapA, fcapB;
string cA, cB, JugA, JugB;
%eden
func repeatChar
(
        auto s, i;
        s = substr("", 1, $2);
        for (i = 1; i <= $2; i++)
                s[i] = $1;
        return s;
)

cA is repeatChar('~', widthA*contentA);
cB is repeatChar('~', widthB*contentB);
JugA is repeatChar('|', 2*capA+2+widthA);
JugB is repeatChar('|', 2*capB+2+widthB);
%scout

fcapA = ([bmenu1.ne+(0, -(2+capA).r), capA, 1],
         [bmenu1.ne+((widthA+1).c, -(2+capA).r), capA, 1],
         [bmenu1.ne+(0, -2.r), 1, widthA+2]);
wcapA = (frame:fcapA, string:JugA);
wcontentA = {
        frame:  ([wcapA.frame.3.nw+(1.c,-contentA.r), contentA, widthA]),
        string: cA,
        bgcolor:"yellow"
};

fcapB = ([fcapA.2.sw+(2.c, -capB.r-1), capB, 1],
         [fcapA.2.sw+((widthB+3).c, -capB.r-1), capB, 1],
         [fcapA.3.ne+(1.c-1, 0), 1, widthB+2]);
wcapB = (frame:fcapB, string:JugB);
wcontentB = {
        frame:  ([wcapB.frame.3.nw+(1.c,-contentB.r), contentB, widthB]),
        string: cB,
        bgcolor:"yellow"
};

window wstatus;
wstatus = {
        frame:  ([fcapB.2.ne+(2.c, (capB/2).r),1,strlen(targ // totstat)]),
        string: targ // totstat
};

#window don;
#point p, q;
#p = (100, 300);
#q = (300, 400);
#don = {
#       type:   DONALD,
#       box:    [p, q],
#       pict:   "view"
#};

screen = < wmenu1 / wmenu2 / wmenu3 / wmenu4 / wmenu5
           / wcontentA / wcontentB / wcapA / wcapB / wstatus >;

integer input;
```

The Jugs Example

D.3. Original Display Specification

```
1     func jugline    /* specifying a line of a jug display */
2     (
3             /* $1 is the line number of the display, $2 is height of jug */
4             /* $3 is the width of the jug, $4 is the content of the jug */
5             /* line number 0 corresponds to the base of the jug */
6             auto c,r,s,t;
7             r = repchar(' ',$3-1);
8             s = repchar(((($1>$4)||($1<0))?' ':(($1==0)?'|':'*')),$3);
9             c = ((($1>$2)||($1<0))?" ":"|");
10            t = r // c // s // c // r;
11            return t;
12    )
13
14    func jugdisplay
15    (
16            /* $1 is height of display, $2 is height of the jug */
17            /* $3 is the width of the jug, $4 is the content of the jug */
18            /* func returns list of strings representing the display of the jug */
19
20            auto s,i;
21            s = [];
22            for (i=$1; i>=0; i--)
23            (
24                    append s,jugline(i-1, $2,$3,$4);
25            )
26            return s;
27    )
28
29    /*
30    proc display : jugA, jugB, menuform, stat, targt
31    (
32            auto s;
33            s = displayright(jugA, displayright(jugB, displayright(targt,stat)));
34            display_list(displayabove(s, menuform));
35    )
36    */
37
38    _display is displayabove(displayright(jugA, displayright(jugB, displayright(targt,sta
      t))), menuform);
39
40    proc display : _display
41    (
42            display_list(_display);
43    )
44
45    proc display_list
46    (
47            /* display a display_list $1 */
48            auto i;
49            for (i=1; i<=$1#; i++)
50            (
51                    writeln($1[i]);
52            )
53    )
54
55    func menudisplay
56    (
57            /* construct display list for menu $1[1] with associated status $1[2] */
58            auto i,s,l;
59            s = "";
60            for (i=1; i<=$1[1]#; i++)
61            (
62                    s = s // "  " // (($1[2][i])?$1[1][i]:("<" // $1[1][i] // ">"));
63            )
64            l = [s];
65            return l;
66    )
67
68
69    func displayabove
70    (
```



```
71            /* two display lists $1 and $2 -> display list for $1 above $2 */
72            auto s,i;
73            s = $1;
74            for (i=1; i<=$2#; i++) (
75                    append s, $2[i];
76            )
77            return s;
78    )
79
80    func displayright
81    (
82            /* two display lists $1 and $2 -> display list for $1 to right of $2 */
83            auto s,i;
84            s = [];
85            for (i=1; i<=$1#; i++) (
86                    append s, ($1[i] // $2[i]);
87            )
88            return s;
89    )
90
91    func messdisplay
92    (
93            /* display text line in a specified vertical position */
94            /* $1+1 is the height  of the display, $2 is the message */
95            auto i,s;
96            s=[];
97            for (i=0; i<=$1; i++) (
98                    append s, ((i==($1/2))?$2:repchar(' ',$2#));
99            )
100           return s;
101   )
102
103
104   func repchar
105   (
106           /* $1=char, $2=number of repetitions */
107           auto s, i;
108           s = substr("", 1, $2);
109           for (i = 1; i <= $2; i++)
110                   s[i] = $1;
111           return s;
112   )
113
114   jugA is jugdisplay(height, capA, widthA, contentA);
115   jugB is jugdisplay(height, capB, widthB, contentB);
116   menuform is menudisplay([menu,menustatus]);
```

Appendix E

The Visualisation Example

E.1. The Script

E.2. Sample Output

The Visualisation Example

E.1. The Script

```
%scout
window don1, don2, arca1, arca2, d1_title, d2_title, a1_title, a2_title;
window d1_label, d2_label, a1_label, a2_label;
point p1 ,q1 ,p2 ,q2, p3, q3, q4,p4;
string d1t, d2t, a1t, d1l, d2l, a1l, a2l;
integer _numcovedge, _sumsum, numgeodesics;
string geolist;
%eden - simulate bridging definition
_sumsum is sumsum[4];
geolist is list_to_str(geodesics);
%scout
p1 = (10, 10);
q1 = (350, 355);
don1 = {
        type:   DONALD,
        box:    [p1, q1],
        pict:   "CONFIG",
        bgcolor:"grey90",
        fgcolor:"brown",
        border: 1
};
d1l = "Arrangement A";
d1_label = {
        frame:  ([don1.box.nw + (1.c, 1.r), 1, strlen(d1l)]),
        string: d1l,
        fgcolor:        "gray1"
};
d1t = itos(_sumsum) // " minimal triangular regions";
d1_title = {
        frame:  ([don1.box.s - (strlen(d1t).c/2, 2.r), 1, strlen(d1t)]),
        string: d1t,
        fgcolor:        "gray1"
};

integer minPx, maxPx, minPy, maxPy;

p3 = (380,10);
q3 = (550,355);
don2 = {
        type:   DONALD,
        box:    [p3, q3],
        pict:   "POSET",
        bgcolor:"grey90",
        fgcolor:"MidnightBlue",
        xmin:   minPx - (maxPx - minPx) / 3,
        xmax:   maxPx + (maxPx - minPx) / 3,
        ymin:   minPy - (maxPy - minPy) / 10,
        ymax:   maxPy + (maxPy - minPy) / 5,
        border: 1
};
d2l = "Poset P'";
d2_label = {
        frame:  ([don2.box.nw + (1.c, 1.r), 1, strlen(d2l)]),
        string: d2l,
        fgcolor:        "gray1"
};
d2t = "\n " // itos(_numcovedge) // " covering edges";
d2_title = {
        frame:  ([don2.box.s - (strlen(d2t).c/2, 1.r), 3, strlen(d2t)]),
        string: d2t,
        fgcolor:        "gray1"
};

p2 = (10, 370);
q2 = (350, 715);
arca1 = {
        type:   ARCA,
        box:    [p2, q2],
        pict:   "view2",
        xmin:   -650,
        ymin:   -650,
```



```
        xmax:   650,
        ymax:   650,
        bgcolor: "grey80",
        border: 1
};
a1l = "Cayley diagram S4";
a1_label = {
        frame:  ([arca1.box.nw + (1.c, 1.r), 1, strlen(a1l)]),
        string: a1l,
        fgcolor:        "gray1"
};
a1t = itos(numgeodesics) // " geodesics " // geolist;
a1_title = {
        frame:  ([arca1.box.s - (strlen(a1t).c/2, 2.r), 1, strlen(a1t)]),
        string: a1t,
        fgcolor:        "gray1"
};

integer minQx, maxQx, minQy, maxQy;

p4 = (380, 370);
q4 = (550, 715);
arca2 = {
        type:   ARCA,
        box:    [p4, q4],
        pict:   "view1",
        bgcolor: "grey80",
        xmin:   minQx - (maxQx - minQx) / 3,
        xmax:   maxQx + (maxQx - minQx) / 3,
        ymin:   minQy - (maxQy - minQy) / 5,
        ymax:   maxQy + (maxQy - minQy) / 5,
        border: 1
};

a2l = "Poset P";
a2_label = {
        frame:  ([arca2.box.sw + (1.c, -2.r), 1, strlen(a2l)]),
        string: a2l,
        fgcolor:        "gray1"
};

# screen = < don1 / don2 / arca1 / arca2 / d1_title / d2_title / a1_title >;
screen = < d1_label / d1_title / d2_label / d2_title /
           a1_label / a1_title / a2_label / don1 / don2 / arca1 / arca2 >;
%donald
# this is the DoNaLD script that defines Figure 1 (a) in the viewport "CONFIG"
viewport CONFIG

real sc,size
point O,Oa,Ob
O,Oa,Ob=(500,500),O-(size,size),O+(size,size)
size,sc = 380.0,2*size
real a12,a23,a34,b12,b23,b34
point A1,A2,A3,A4,B1,B2,B3,B4
line l1,l2,l3,l4
# a12 and b12 determine the distances between the L and R ends of lines 1 and 2
a12=1.0
a23=5.0
a34=2.0
b12=2.0
b23=5.0
b34=1.0
# A1 and B1 are the left and right endpoints of line 1
A1=Oa
A2=Oa+(0, a12 div (a12+a23+a34))*sc
A3=Oa+(0,(a12+a23) div (a12+a23+a34))*sc
A4=Oa+(0,(a12+a23+a34) div (a12+a23+a34))*sc
B1=Ob
B2=Ob-(0,(b12) div (b12+b23+b34))*sc
B3=Ob-(0,(b12+b23) div (b12+b23+b34))*sc
B4=Ob-(0,(b12+b23+b34) div (b12+b23+b34))*sc
```

```
143   l1=[A1,B1]
144   l2=[A2,B2]
145   l3=[A3,B3]
146   l4=[A4,B4]
147
148   label j1,j2,j3,j4,k1,k2,k3,k4
149   j1 = label("1",A1)
150   j2 = label("2",A2)
151   j3 = label("3",A3)
152   j4 = label("4",A4)
153   k1 = label("1",B1)
154   k2 = label("2",B2)
155   k3 = label("3",B3)
156   k4 = label("4",B4)
157
158   # this is the DoNaLD script that defines Figure 1(b) in the viewport "POSET"
159   viewport POSET
160
161   # r12 determines the LR position of the line 1,2 intersection
162   # x12 is to be the crossing index of the line 1,2 intersection
163   #     eg this is 1 if lines 1 and 2 are the top pair at their pt of Xn
164   real r12,r23,r34,r13,r24,r14
165   int x12,x23,x34,x13,x24,x14
166   r12,r23,r34 = a12 div b12 , a23 div b23, a34 div b34
167   r13 = (a12+a23) div (b12+b23)
168   r24 = (a23+a34) div (b23+b34)
169   r14 = (a12+a23+a34) div (b12+b23+b34)
170
171   # z123 = 1 if line 1 crosses line 2 before line 3 crosses line 2 in LR order
172   int z123,z124,z134,z234
173   int Z123,Z124,Z134,Z234
174   z123 = if r12<r23 then 1 else 0
175   z124 = if r12<r24 then 1 else 0
176   z134 = if r13<r34 then 1 else 0
177   z234 = if r23<r34 then 1 else 0
178   Z123 = if r13<r12 then 1 else 0
179   Z124 = if r14<r12 then 1 else 0
180   Z134 = if r14<r13 then 1 else 0
181   Z234 = if r24<r23 then 1 else 0
182
183   # x12 calcs the crossing index for lines 1 and 2
184   # "by default" this is 1 but is incremented if line 3 or 4 crosses line 1
185   # before line 2 crosses line 1 in LR order
186   x12 = 1+Z123+Z124
187   x13 = 1+(1-Z123)+Z134
188   x14 = 1+(1-Z124)+(1-Z134)
189   x23 = 2-z123+Z234
190   x24 = 2-z124+(1-Z234)
191   x34 = 3-z134-z234
192
193   # these are the points of the poset of intersections
194   # v is a vertical, m a global magnification factor
195   int v,m
196   v,m=8,50
197   point orig,p12,p23,p34,p13,p24,p14
198   p12 = orig+(x12,r12*v)*m
199   p23 = orig+(x23,r23*v)*m
200   p24 = orig+(x24,r24*v)*m
201   p34 = orig+(x34,r34*v)*m
202   p14 = orig+(x14,r14*v)*m
203   p13 = orig+(x13,r13*v)*m
204   orig = (400,400)
205   # Line l1213 occurs in the poset if the intersection of lines 1 and 2
206   # and the intersection of lines 1 and 3 corresponds to a covering edge
207   # Line l1213 is present if the parameter d1213 evaluates to 1
208   # it otherwise contracts to the origin
209   line l1213,l1214,l1314,l2324,l1223,l1224,l1334,l2334,l1323,l1424,l2434,l1434
210   int d1213,d1214,d1314,d2324,d1223,d1224,d1334,d2334,d1323,d1424,d2434,d1434
211   l1214 = [p12*d1214,p14*d1214]
212   l1323 = [p13*d1323,p23*d1323]
213   l1224 = [p12*d1224,p24*d1224]
```



```
214   l1424 = [p14*d1424,p24*d1424]
215   l2324 = [p23*d2324,p24*d2324]
216   l1334 = [p13*d1334,p34*d1334]
217   l1213 = [p12*d1213,p13*d1213]
218   l1434 = [p14*d1434,p34*d1434]
219   l1314 = [p13*d1314,p14*d1314]
220   l1223 = [p12*d1223,p23*d1223]
221   l2434 = [p24*d2434,p34*d2434]
222   l2334 = [p23*d2334,p34*d2334]
223
224   # d1213 is 1 if the crossing index of lines 1 and 2 differs from that of
225   # lines 1 and 3  and line 4 doesn't cross line 1 between its points
226   # of intersection with lines 2 and 3
227   d1334 = if !(x13==x34) && ((r23-r13)*(r23-r34)>0) then 1 else 0
228   d2334 = if !(x23==x34) && ((r13-r23)*(r13-r34)>0) then 1 else 0
229   d1224 = if !(x12==x24) && ((r23-r12)*(r23-r24)>0) then 1 else 0
230   d1223 = if !(x12==x23) && ((r24-r12)*(r24-r23)>0) then 1 else 0
231   d1323 = if !(x13==x23) && ((r34-r13)*(r34-r23)>0) then 1 else 0
232   d1424 = if !(x14==x24) && ((r34-r14)*(r34-r24)>0) then 1 else 0
233   d2434 = if !(x24==x34) && ((r14-r24)*(r14-r34)>0) then 1 else 0
234   d1434 = if !(x14==x34) && ((r24-r14)*(r24-r34)>0) then 1 else 0
235   d1213 = if !(x12==x13) && ((r14-r12)*(r14-r13)>0) then 1 else 0
236   d1214 = if !(x12==x14) && ((r13-r12)*(r13-r14)>0) then 1 else 0
237   d1314 = if !(x13==x14) && ((r12-r13)*(r12-r14)>0) then 1 else 0
238   d2324 = if !(x23==x24) && ((r12-r23)*(r12-r24)>0) then 1 else 0
239
240   int numcovedge
241   numcovedge = d1213+d1223+d1224+d1214+d1314+d1334+d1323+d2324+d2334+d2434+d1434+d1424
242
243   int g1213,g1214,g1314,g2324,g1223,g1224,g1334,g2334,g1323,g1424,g2434,g1434
244   g1213 = if (((p12.1-p13.1)*(p12.2-p13.2)) < 0) then 0 else 1
245   g1214 = if (((p12.1-p14.1)*(p12.2-p14.2)) < 0) then 0 else 1
246   g1314 = if (((p13.1-p14.1)*(p13.2-p14.2)) < 0) then 0 else 1
247   g2324 = if (((p23.1-p24.1)*(p23.2-p24.2)) < 0) then 0 else 1
248   g1223 = if (((p12.1-p23.1)*(p12.2-p23.2)) < 0) then 0 else 1
249   g1224 = if (((p12.1-p24.1)*(p12.2-p24.2)) < 0) then 0 else 1
250   g1334 = if (((p13.1-p34.1)*(p13.2-p34.2)) < 0) then 0 else 1
251   g2334 = if (((p23.1-p34.1)*(p23.2-p34.2)) < 0) then 0 else 1
252   g1323 = if (((p13.1-p23.1)*(p13.2-p23.2)) < 0) then 0 else 1
253   g1424 = if (((p14.1-p24.1)*(p14.2-p24.2)) < 0) then 0 else 1
254   g2434 = if (((p24.1-p34.1)*(p24.2-p34.2)) < 0) then 0 else 1
255   g1434 = if (((p14.1-p34.1)*(p14.2-p34.2)) < 0) then 0 else 1
256
257
258   int r1213,r1214,r1314,r2324,r1223,r1224,r1334,r2334,r1323,r1424,r2434,r1434
259   r1213 = if ((r12-r13) < 0) then 0 else 1
260   r1214 = if ((r12-r14) < 0) then 0 else 1
261   r1314 = if ((r13-r14) < 0) then 0 else 1
262   r2324 = if ((r23-r24) < 0) then 0 else 1
263   r1223 = if ((r12-r23) < 0) then 0 else 1
264   r1224 = if ((r12-r24) < 0) then 0 else 1
265   r1334 = if ((r13-r34) < 0) then 0 else 1
266   r2334 = if ((r23-r34) < 0) then 0 else 1
267   r1323 = if ((r13-r23) < 0) then 0 else 1
268   r1424 = if ((r14-r24) < 0) then 0 else 1
269   r2434 = if ((r24-r34) < 0) then 0 else 1
270   r1434 = if ((r14-r34) < 0) then 0 else 1
271
272   int U1213,U1214,U1314,U2324,U1223,U1224,U1334,U2334,U1323,U1424,U2434,U1434
273   U1213 = g1213 * r1213 * d1213
274   U1214 = g1214 * r1214 * d1214
275   U1314 = g1314 * r1314 * d1314
276   U2324 = g2324 * r2324 * d2324
277   U1223 = g1223 * r1223 * d1223
278   U1224 = g1224 * r1224 * d1224
279   U1334 = g1334 * r1334 * d1334
280   U2334 = g2334 * r2334 * d2334
281   U1323 = g1323 * r1323 * d1323
282   U1424 = g1424 * r1424 * d1424
283   U1434 = g1434 * r1434 * d1434
284   U2434 = g2434 * r2434 * d2434
```

```
285
286   int u1213,u1214,u1314,u2324,u1223,u1224,u1334,u2334,u1323,u1424,u2434,u1434
287   u1213 = g1213 * (1-r1213) * d1213
288   u1214 = g1214 * (1-r1214) * d1214
289   u1314 = g1314 * (1-r1314) * d1314
290   u2324 = g2324 * (1-r2324) * d2324
291   u1223 = g1223 * (1-r1223) * d1223
292   u1224 = g1224 * (1-r1224) * d1224
293   u1334 = g1334 * (1-r1334) * d1334
294   u2334 = g2334 * (1-r2334) * d2334
295   u1323 = g1323 * (1-r1323) * d1323
296   u1424 = g1424 * (1-r1424) * d1424
297   u1434 = g1434 * (1-r1434) * d1434
298   u2434 = g2434 * (1-r2434) * d2434
299
300    int v1213,v1214,v1314,v2324,v1223,v1224,v1334,v2334,v1323,v1424,v2434,v1434
301    v1213 = (1-g1213) * (1-r1213) * d1213
302    v1214 = (1-g1214) * (1-r1214) * d1214
303    v1314 = (1-g1314) * (1-r1314) * d1314
304    v2324 = (1-g2324) * (1-r2324) * d2324
305    v1223 = (1-g1223) * (1-r1223) * d1223
306    v1224 = (1-g1224) * (1-r1224) * d1224
307    v1334 = (1-g1334) * (1-r1334) * d1334
308    v2334 = (1-g2334) * (1-r2334) * d2334
309    v1323 = (1-g1323) * (1-r1323) * d1323
310    v1424 = (1-g1424) * (1-r1424) * d1424
311    v1434 = (1-g1434) * (1-r1434) * d1434
312    v2434 = (1-g2434) * (1-r2434) * d2434
313
314
315    int V1213,V1214,V1314,V2324,V1223,V1224,V1334,V2334,V1323,V1424,V2434,V1434
316    V1213 = (1-g1213) * r1213 * d1213
317    V1214 = (1-g1214) * r1214 * d1214
318    V1314 = (1-g1314) * r1314 * d1314
319    V2324 = (1-g2324) * r2324 * d2324
320    V1223 = (1-g1223) * r1223 * d1223
321    V1224 = (1-g1224) * r1224 * d1224
322    V1334 = (1-g1334) * r1334 * d1334
323    V2334 = (1-g2334) * r2334 * d2334
324    V1323 = (1-g1323) * r1323 * d1323
325    V1424 = (1-g1424) * r1424 * d1424
326    V1434 = (1-g1434) * r1434 * d1434
327    V2434 = (1-g2434) * r2434 * d2434
328
329
330   %donald
331   int in12, in23, in24, in34, in14, in13
332   in13 = U1314+U1334+U1323+u1213+v1213+V1334+V1323+V1314
333   in23 = U2324+U2334+u1223+u1323+v1323+v1223+V2324+V2334
334   in24 = U2434+u2324+u1224+u1424+V2434+v2324+v1224+v1424
335   in34 = u1334+u2334+u1434+u2434+v1334+v2334+v1434+v2434
336   in14 = U1424+U1434+u1214+u1314+V1424+V1434+v1214+v1314
337   in12 = U1213+U1214+U1223+U1224+V1224+V1223+V1213+V1214
338
339   # Arca diagram poset has 6 vertices respectively representing
340   # intersections between line pairs
341   # 12, 23, 24, 34, 14, 13
342
343   %arca
344   mode poset = 'ab'-diag 6
345   for int 0 : i = 1 to 6 do
346           mode poset!i = abst vert 2
347   od
348   mode a_poset = col 6
349   mode b_poset = col 6
350   for int 0: i = 1 to 6 do
351           mode a_poset(i) = int 0
352           mode b_poset(i) = int 0
353   od
354   mode h = int 0
355   mode v = int 0
```



```
356   v = h
357   h = 200
358
359   a_poset(1) = (DONALD(u1213)*6)+(DONALD(u1214)*5)+(DONALD(u1223)*2)+(DONALD(u1224)*3)
360   a_poset(2) = (DONALD(u2324)*3)+(DONALD(u2334)*4)+(DONALD(U1223)*1)+(DONALD(U1323)*6)
361   a_poset(3) = (DONALD(U2324)*2)+(DONALD(u2434)*4)+(DONALD(U1224)*1)+(DONALD(U1424)*5)
362   a_poset(4) = (DONALD(U2334)*2)+(DONALD(U2434)*3)+(DONALD(U1334)*6)+(DONALD(U1434)*5)
363   a_poset(5) = (DONALD(u1424)*3)+(DONALD(u1434)*4)+(DONALD(U1214)*1)+(DONALD(U1314)*6)
364   a_poset(6) = (DONALD(u1323)*2)+(DONALD(u1334)*4)+(DONALD(u1314)*5)+(DONALD(U1213)*1)
365
366   b_poset(1) = (DONALD(v1213)*6)+(DONALD(v1214)*5)+(DONALD(v1223)*2)+(DONALD(v1224)*3)
367   b_poset(2) = (DONALD(v2324)*3)+(DONALD(v2334)*4)+(DONALD(V1223)*1)+(DONALD(V1323)*6)
368   b_poset(3) = (DONALD(V2324)*2)+(DONALD(v2434)*4)+(DONALD(V1224)*1)+(DONALD(V1424)*5)
369   b_poset(4) = (DONALD(V2334)*2)+(DONALD(V2434)*3)+(DONALD(V1334)*6)+(DONALD(V1434)*5)
370   b_poset(5) = (DONALD(v1424)*3)+(DONALD(v1434)*4)+(DONALD(V1214)*1)+(DONALD(V1314)*6)
371   b_poset(6) = (DONALD(v1323)*2)+(DONALD(v1334)*4)+(DONALD(v1314)*5)+(DONALD(V1213)*1)
372
373   mode prodba = abst col
374   prodba = b_poset . a_poset
375   mode prodab = abst col
376   prodab = a_poset . b_poset
377
378   mode sumsum = int 0
379   mode sum = vert 6
380   mode pabi = vert 6
381   mode pbai = vert 6
382   mode upabi = vert 6
383   mode upbai = vert 6
384   mode eqabbai = vert 6
385   for int 0: i = 1 to 6 do
386           mode sum[i] = int 0
387           mode pabi[i] = int 0
388           mode pbai[i] = int 0
389           mode upabi[i] = int 0
390           mode upbai[i] = int 0
391           mode eqabbai[i] = int 0
392   od
393
394   for int 6: i = 1 to 6 do
395           sum[i] =  (1-upabi[i]*upbai[i])*(1-eqabbai[i])
396           pbai[i] = prodba(i)
397           pabi[i] = prodab(i)
398           upabi[i] = if (pabi[i]==UNDEF) 1 else 0
399           upbai[i] = if (pbai[i]==UNDEF) 1 else 0
400           eqabbai[i] = if (pabi[i]==pbai[i]) 1 else 0
401   od
402
403   sumsum = sum[1]+sum[2]+sum[3]+sum[4]+sum[5]+sum[6]
404
405   mode start = int 0
406   start =1
407
408   #poset!1 = [ DONALD(x12)*h, dist(start -> 1 , a_poset, b_poset)*v]
409   #poset!2 = [ DONALD(x23)*h, dist(start -> 2 , a_poset, b_poset)*v]
410   #poset!3 = [ DONALD(x24)*h, dist(start -> 3 , a_poset, b_poset)*v]
411   #poset!4 = [ DONALD(x34)*h, dist(start -> 4 , a_poset, b_poset)*v]
412   #poset!5 = [ DONALD(x14)*h, dist(start -> 5 , a_poset, b_poset)*v]
413   #poset!6 = [ DONALD(x13)*h, dist(start -> 6 , a_poset, b_poset)*v]
414
415
416   %donald
417
418   int ixa, ixb, ixc
419   ixa = (if ((in12==0)&&(x12==1)) then 1 else 0)*1+(if ((in23==0)&&(x23==1)) then 1 els
      e 0)*2+(if ((in24==0)&&(x24==1)) then 1 else 0)*3+(if ((in34==0)&&(x34==1)) then 1 els
      e 0)*4+(if ((in14==0)&&(x14==1)) then 1 else 0)*5+(if ((in13==0)&&(x13==1)) then 1 els
      e 0)*6
420   ixb = (if ((in12==0)&&(x12==2)) then 1 else 0)*1+(if ((in23==0)&&(x23==2)) then 1 els
      e 0)*2+(if ((in24==0)&&(x24==2)) then 1 else 0)*3+(if ((in34==0)&&(x34==2)) then 1 els
      e 0)*4+(if ((in14==0)&&(x14==2)) then 1 else 0)*5+(if ((in13==0)&&(x13==2)) then 1 els
      e 0)*6
```

```
421   ixc = (if ((in12==0)&&(x12==3)) then 1 else 0)*1+(if ((in23==0)&&(x23==3)) then 1 els
      e 0)*2+(if ((in24==0)&&(x24==3)) then 1 else 0)*3+(if ((in34==0)&&(x34==3)) then 1 els
      e 0)*4+(if ((in14==0)&&(x14==3)) then 1 else 0)*5+(if ((in13==0)&&(x13==3)) then 1 els
      e 0)*6
422
423   %arca
424   mode ap = abst col 10
425   mode bp = abst col 10
426   mode cp = abst col 10
427
428   ap = (10,7, (DONALD(ixa)))
429   bp = (10,8, (DONALD(ixb)))
430   cp = (10,9, (DONALD(ixc)))
431
432   mode apex = abst col 10
433   apex = a_poset :: (0,0,0,0)
434   mode bpex = abst col 10
435   bpex = b_poset :: (0,0,0,0)
436
437   # start = 10 is a mistake at this point! - it results in a re-evaluation of
438   # poset!1 with disastrous results
439
440   poset!1 = [ DONALD(x12)*h, dist(start -> 1 , apex, bpex, ap, bp, cp)*v - 2*v]
441   poset!2 = [ DONALD(x23)*h, dist(start -> 2 , apex, bpex, ap, bp, cp)*v - 2*v]
442   poset!3 = [ DONALD(x24)*h, dist(start -> 3 , apex, bpex, ap, bp, cp)*v - 2*v]
443   poset!4 = [ DONALD(x34)*h, dist(start -> 4 , apex, bpex, ap, bp, cp)*v - 2*v]
444   poset!5 = [ DONALD(x14)*h, dist(start -> 5 , apex, bpex, ap, bp, cp)*v - 2*v]
445   poset!6 = [ DONALD(x13)*h, dist(start -> 6 , apex, bpex, ap, bp, cp)*v - 2*v]
446
447
448   start = 10
449
450
451   mode k = abst vert 4
452   mode unit = int 0
453   mode l = int 0
454   mode s4 = abst 'abc'-diag
455
456   mode dia = 'ab'-diag 4
457   mode dib = abst 'ab'-diag 4
458   mode dic = abst 'ab'-diag 4
459   mode did = abst 'ab'-diag 4
460   mode die = abst 'ab'-diag 4
461   mode dif = abst 'ab'-diag 4
462
463   mode dia!1 = abst vert 3
464   mode dia!2 = abst vert 3
465   mode dia!3 = abst vert 3
466   mode dia!4 = abst vert 3
467
468   mode point = abst vert
469
470   k = [25, 25, 25, 25]
471   l = 50
472
473   unit= 8
474
475   point = [0, 0, 0-l * unit]
476   dia!1 = point + [0, k[1]*unit,0]
477   dia!2 = point + [k[2]*unit, 0,0]
478   dia!3 = point - [0,k[3]*unit,0]
479   dia!4 = point - [k[4]*unit, 0,0]
480
481   mode a_s4 = abst col
482   mode b_s4 = abst col
483   mode c_s4 = abst col
484   mode d_s4 = abst col
485   mode a_dia = abst col
486   mode b_dia = abst col
487
488   a_dia = (1,2)$(3,4)
```



```
489   b_dia = (1,4)$(2,3)
490
491   a_s4 = b_dia :: a_dia :: b_dia :: a_dia :: a_dia :: a_dia
492   b_s4 = a_dia :: b_dia :: a_dia :: b_dia :: b_dia :: b_dia
493   c_s4 = (2,8)$(6,12)$(14,4)$(16,10)$(1,19)$(3,21)$(5,18)$(7,22)$(9,17)$(11,23)$(13,20)
      $(15,24)
494   dib = rot( dia, [90,1,3])
495   dic = rot( dia, [180,1,3])
496   did = rot( dia, [270,1,3])
497   die = rot( dia, [90,2,3])
498   dif = rot( dia, [270,2,3])
499   mode angle = int 0
500   angle = 45
501
502   s4 =  rot(dia::dib::dic::did::die::dif, [angle,2,3])
503
504   mode S4 = 'abc'-diag 24
505   for int 0 : i = 1 to 24 do
506           mode S4!i = vert 2
507           mode S4!i[1] = int 0
508           mode S4!i[2] = int 0
509           S4!i[1] = s4!i[1]
510           S4!i[2] =  -450 + dist(1 -> i ,a_s4 ,b_s4 ,c_s4) *150
511   od
512
513   S4!15[1] = -500
514   S4!13[1] = -300
515   S4!17[1] = -100
516   S4!21[1] = 100
517   S4!7[1] = 300
518   S4!5[1] = 500
519
520   mode a_S4 = abst col
521   mode b_S4 = abst col
522   mode c_S4 = abst col
523
524   a_S4 = a_s4
525   b_S4 = b_s4
526   c_S4 = c_s4
527
528   mode ht = vert 6
529   for int 0 : i = 1 to 6 do
530           mode ht[i] = int 0
531           ht[i] = dist(start->i, apex, bpex, ap, bp, cp) - 2
532   od
533
534   %eden
535   func inv_image (
536           auto i, result;
537           result = [];
538           for (i=1; i<=$1#; i++) {
539                   if ($1[i]==$2) result = result // [i];
540           }
541           return result;
542   }
543
544   func prod (
545           auto i, result;
546           result = 1;
547           for (i=1; i<=$1#; i++) {
548                   result = result*$1[i];
549           }
550           return result;
551   }
552
553   func maplen (
554           auto i, result;
555           result = [];
556           for (i=1; i<=$1#; i++) {
557                   result = result // [$1[i]#];
558           }
```

```
559             return result;
560     )
561
562     func max (
563             auto i, result;
564             result = 0;
565             for (i=1; i<=$1#; i++) {
566                     if ($1[i] != @ && result < $1[i]) result = $1[i];
567             }
568             return result;
569     )
570
571     func mapinv_im (
572             auto i, result;
573             result= [];
574             for (i=0; i<=$2; i++) {
575                     result = result // [inv_image($1,i)];
576             }
577             return result;
578     )
579
580     func vert_to_list (
581             auto i, result;
582             result= [];
583             for (i=1; i<=$1[4]#; i++) {
584                     result = result // [$1[4][i][4]];
585             }
586             return result;
587     )
588
589     listht is vert_to_list(ht);
590     numgeodesics is prod(maplen(atht));
591     atht is mapinv_im(listht,max(listht));
592
593     func rank (
594             auto code;
595             code = [_x12,_x23,_x24,_x34,_x14,_x13];
596             return code[$1];
597     )
598
599     func maprank (
600             auto i, result;
601             result= [];
602             for (i=1; i<=$1#; i++) {
603                     if ($1[i]#==1) result = result // [[rank($1[i][1])]];
604                     if ($1[i]#==2) result = result // [[rank($1[i][1]),rank($1[i][2])]];
605             }
606             return result;
607     )
608
609     func pairappend (
610             auto result;
611             if ($2#==1) result = [$1 // $2];
612             if ($2#==2) result = [$1 // $2, $1 // [$2[2], $2[1]]] ;
613             return result;
614     )
615
616
617     func treeappend (
618             auto result, result2, i,j;
619             result = [[]];
620             for (i=1; i<=$1#; i++) {
621                     result2 = [];
622                     for (j=1; j<=result#; j++) {
623                             result2 = result2 // pairappend(result[j],$1[i]);
624                     }
625                     result = result2;
626             }
627             return result;
628     )
629
```



```
630     func list_to_str (
631             para list;
632             auto result, result2, i, j;
633             result = "<";
634             for (i=1; i<=list#; i++) {
635                     result2 = "";
636                     for (j=1; j<=list[i]#; j++)
637                             result2 = result2 // str(list[i][j]);
638                     if (i == list#)
639                             result = result // result2 // ">";
640                     else
641                             result = result // result2 // ",";
642             }
643             return result;
644     )
645
646     ranklist is maprank(atht);
647     geodesics is treeappend(ranklist);
648
649     func indices (
650             auto res, i, collist;
651             collist = [vert_to_list(s4_a), vert_to_list(s4_c), vert_to_list(s4_b)];
652             res = [1];
653             for (i=1; i<=$1#; i++) {
654                     if ($1[i]#==1) res = res // [collist[$1[i][1]][res[res#]]];
655                     if ($1[i]#==2) res = res // [collist[$1[i][1]][res[res#]]]
656                                                 // [collist[$1[i][2]][res[res#]]]
657                             // [collist[$1[i][2]][collist[$1[i][1]][res[res#]]]];
658             }
659             return res;
660     )
661
662     func paths (
663             auto res, i, collist;
664             collist = [vert_to_list(s4_a), vert_to_list(s4_c), vert_to_list(s4_b)];
665             res = [1];
666             for (i=1; i<=$1#; i++) {
667                     if ($1[i]#==1) res = res // [collist[$1[i][1]][res[res#]]];
668                     if ($1[i]#==2) res = res
669                             // [collist[$1[i][2]][collist[$1[i][1]][res[res#]]]];
670             }
671             return res;
672     )
673
674     subdiagS4 is indices(ranklist);
675     geotrace is paths(ranklist);
676
677     %donald
678     viewport POSET
679     label 112,123,134,113,114,124
680     boolean labelon
681     labelon = false
682     112 = if labelon then label("12",p12) else label("",p12)
683     113 = if labelon then label("13",p13) else label("",p13)
684     114 = if labelon then label("14",p14) else label("",p14)
685     123 = if labelon then label("23",p23) else label("",p23)
686     124 = if labelon then label("24",p24) else label("",p24)
687     134 = if labelon then label("34",p34) else label("",p34)
688
689     # nice picture if a12 = a23 = 4 a34= 2 b12, b23, b34 = 2.0,5.0,1.0
690     # m = 100 labelon = true v=8
691     # orig = (300, -500)
692     # also get good result with these parameter settings with the original
693     # values of a12, a23, a34 viz 1,5,2
694
695
696     orig = (300,-500)
697     m = 100
698     labelon = true
699
700     %arca
```

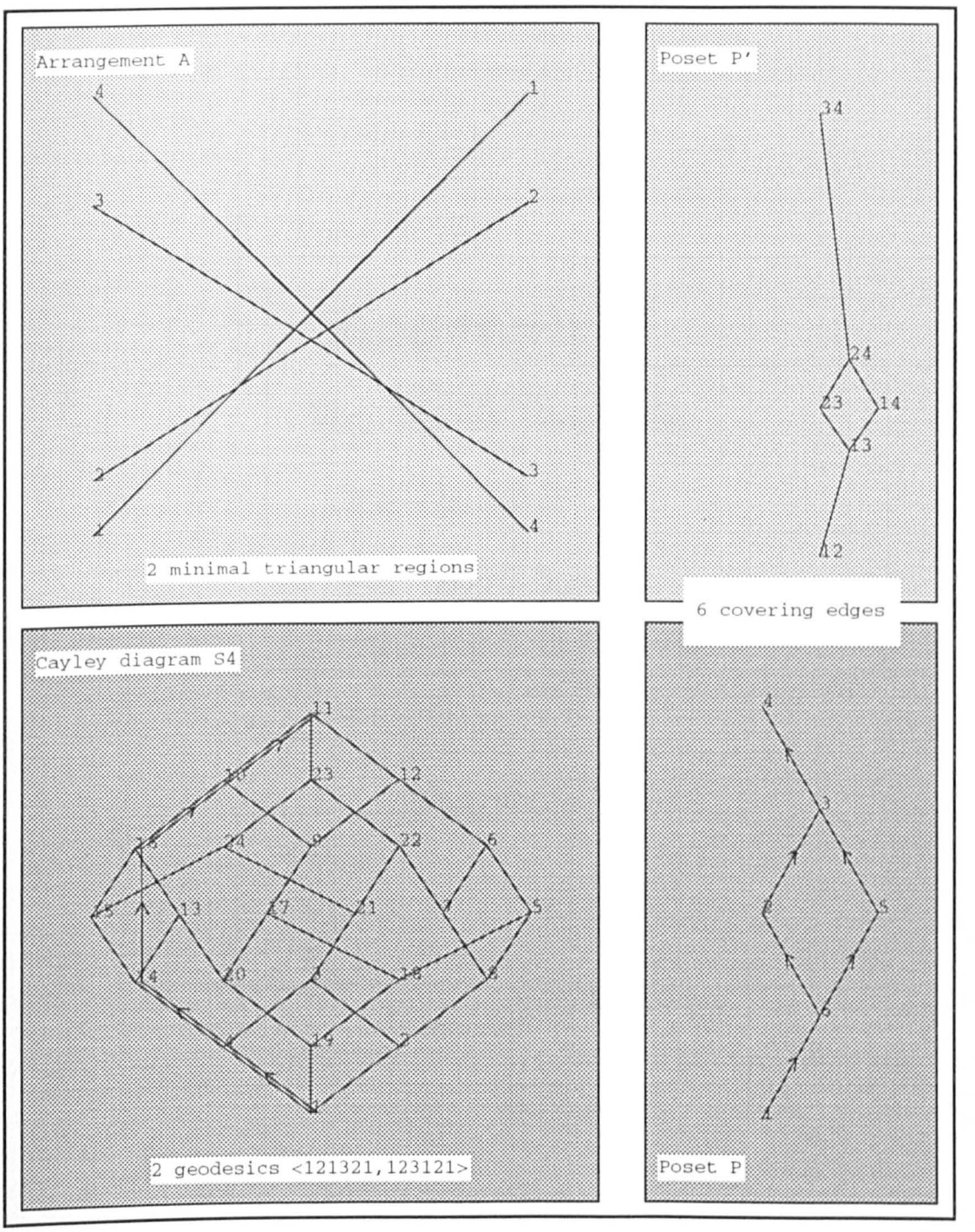
```
701
702   mode path =  'd'-diag 24
703   mode offset = int 0
704   offset = 15
705   for int 0: i = 1 to 24 do
706           mode path!i = abst vert
707           path!i = S4!i + [offset,0]
708   od
709
710   mode d_path = abst col
711   d_path = (24,0)
712
713   %eden
714
715   proc asgndcol: geotrace (
716           auto j;
717           for (j=1; j<=24;j++) path_d[4][j][4] = 0;
718           for (j=1; j<geotrace#; j++) {
719                   path_d[4][geotrace[j]][4] =  geotrace[j+1];
720           }
721   )
722
723   %eden
724   func minpt
725   {
726           para ptlst;
727           auto result;
728           result = [0,0];
729           while (ptlst != []) {
730                   result = [(ptlst[1][2] != @ && ptlst[1][2] < result[1])
731                                   ? ptlst[1][2] : result[1],
732                             (ptlst[1][3] != @ && ptlst[1][3] < result[2])
733                                   ? ptlst[1][3] : result[2]];
734                   shift ptlst;
735           }
736           return result;
737   }
738   func maxpt
739   {
740           para ptlst;
741           auto result;
742           result = [0,0];
743           while (ptlst != []) {
744                   result = [(ptlst[1][2] != @ && ptlst[1][2] > result[1])
745                                   ? ptlst[1][2] : result[1],
746                             (ptlst[1][3] != @ && ptlst[1][3] > result[2])
747                                   ? ptlst[1][3] : result[2]];
748                   shift ptlst;
749           }
750           return result;
751   }
752   minPx is minpt([_p12, _p23, _p24, _p34, _p14, _p13])[1];
753   minPy is minpt([_p12, _p23, _p24, _p34, _p14, _p13])[2];
754   maxPx is maxpt([_p12, _p23, _p24, _p34, _p14, _p13])[1];
755   maxPy is maxpt([_p12, _p23, _p24, _p34, _p14, _p13])[2];
756
757   func apttodpt {
758           para apt;
759           return ['C', apt[4][1][4], apt[4][2][4]];
760   };
761
762   minQx is minpt([apttodpt(poset_1), apttodpt(poset_2), apttodpt(poset_3),
763           apttodpt(poset_4), apttodpt(poset_5), apttodpt(poset_6)])[1];
764   minQy is minpt([apttodpt(poset_1), apttodpt(poset_2), apttodpt(poset_3),
765           apttodpt(poset_4), apttodpt(poset_5), apttodpt(poset_6)])[2];
766   maxQx is maxpt([apttodpt(poset_1), apttodpt(poset_2), apttodpt(poset_3),
767           apttodpt(poset_4), apttodpt(poset_5), apttodpt(poset_6)])[1];
768   maxQy is maxpt([apttodpt(poset_1), apttodpt(poset_2), apttodpt(poset_3),
769           apttodpt(poset_4), apttodpt(poset_5), apttodpt(poset_6)])[2];
770
771   %arca
```



```
772   display 'abc'-S4 on view2 with labels
773   display 'ab'-poset on view1 with labels
774   display 'd'-path on view2
```

The Visualisation Example

E.2. Sample Output


Arrangement A
4
1
3
2
2
1
3
4
2 minimal triangular regions
Poset P'
34
24
23
14
13
12
6 covering edges
Cayley diagram S4
11
23
12
22
13
17
21
20
19
2 geodesics <121321, 123121>
Poset P
4
3
2
1

Appendix F

The Room Example

F.1. The Script

F.2. Sample Output

The Room Example

F.1. The Script

```
%donald

/*
 * The original DoNaLD specification for a room by Edward Yung
 */

viewport view
?prohibit_touch = ON;
#       The following definitions define a room ...
#       ... and some objects inside the room.
#

# The room is rectangle in shape.

int     width, length           # the dimension of room
point   NW, NE, SW, SE          # the 4 corners of room
line    N1, N2, S, E, W         # the 4 walls of room
                                # North wall composites of 2 walls & a door

                                ### Every room has a door ###
openshape door                  # declare the door

within door {
        point   hinge, lock     # a door has a hinge, and a lock
        line    door            # the door itself
        int     width           # the size of the door
        boolean open            # use it as a flag, telling whether ...
                                # ... the door has opened
        door = [hinge, lock]

        lock = hinge + (if open then (0, -width) else (width, 0))
}

within door {
        hinge = ~/NW + (15, -10)        # set the coordinate of the hinge
        open = true                     # the door has opened
        width = 200                     # the size of the door
}

N1 = [NW, (door/hinge.1, NW.2)]
N2 = [(door/hinge.1+door/width, NW.2), NE]
S = [SW, SE]
W = [NW, SW]
E = [NE, SE]

SW = (100, 100)
SE = SW + (width, 0)            # The other 3 corners are relative
NE = SW + (width, length)       # to South-West corner.
NW = SW + (0, length)           #

width, length = 800, 800        # 800*800 pts

################################################################################
openshape table                 # There is a table inside the room.

within table {
        int     width, length           # the dimension of table
        point   NW, NE, SW, SE          # the 4 corners of table
        line    N, S, E, W              # the 4 sides of table

        N = [NW, NE]
        S = [SW, SE]
        W = [NW, SW]
        E = [NE, SE]

        SE = SW + (width, 0)            # the other 3 corners are
        NE = SW + (width, length)       # relative to SW corner
        NW = SW + (0, length)           #

        ##### table/lamp #####
```



```
        openshape lamp                  # There is a lamp on the table
        within lamp {
                point   centre          # center of the table
                int     size            # size of the lamp
                int     half            # half of size
                circle  base

                size = 50
                half = size div 2
                centre = (~/SW + ~/NE) div 2    # at the centre of table

                                        # L1-L8 forms an octagon.
                line    L1, L2, L3, L4, L5, L6, L7, L8

                L1 = [centre + (size, -half), centre + (size, half)]
                L2 = [L1.2, centre + (half, size)]
                L3 = [L2.2, centre + (-half, size)]
                L4 = [L3.2, centre + (-size, half)]
                L5 = [L4.2, centre + (-size, -half)]
                L6 = [L5.2, centre + (-half, -size)]
                L7 = [L6.2, centre + (half, -size)]
                L8 = [L7.2, L1.1]
                base = circle(centre, size * 1.25)
        }
}

within table {
                # let the size of table be 300*300 pts.
        width, length = 300, 300

                # place the SW of table at the center of the room.
        SW = (~/SW + ~/NE) div 2
}

################################################################################
point   plug
point   plug1, plug2            # there are 2 plugs.
plug1 = (S.1+S.2) div 2         # middle of South wall
plug2 = (E.1+E.2) div 2         # middle of East wall

line    cable                   # cable connects the table/lamp and the plug
int     cablelength
cable = [plug, table/lamp/centre]
plug = plug1                    # let cable connects to plug1
cablelength = 600
#
# Now set the line style of cable be dotted line.
#   Because many Donald commands hasn't implemented,
#       I just directly access Eden using the non-standard '?' command.
? A_cable = "linestyle=dashed,dash=13";


################################################################################

openshape desk                  # there is a desk in the room

within desk {
        int     width, length           # the size of the desk
        point   NW, NE, SW, SE          # the 4 corners of the desk
        line    N, S, E, W              # the 4 edges of the desk

        N = [NW, NE]
        S = [SW, SE]
        W = [NW, SW]
        E = [NE, SE]

        width, length = 250, 350

                # initially the desk is placed at (100,100) of the room.
        SW = ~/SW + (15, 15)
        SE = SW + (width, 0)            # the other 3 corners are ...
```

```
143           NW = SW + (0, length)           # ... relative to SW corner
144           NE = NW + (width, 0)            #
145
146           ##### desk/drawer #####
147
148           openshape drawer        # the desk has a drawer
149
150           within drawer {
151                   int     k       # k --  a parameter describes
152                                   #       the condition of drawer
153                                   #               1 --> closed
154                                   #               larger k --> more open
155
156                   int     width, length   # the size of drawer
157                   point   NW, NE, SW, SE  # the 4 corners of the drawer
158                   line    N, S, W, E      # the 4 edges of the drawer
159
160                           # the size of the drawer is always a ratio to ...
161                           # ... the desk.
162                   width = ~/length div 3
163                   length = ~/width - ~/width div k
164                   k = 2                   # initially, the draw is half open
165
166                   NW = ~/NE               # NW is always at desk's NE
167                   SW = NW - (0, width)    # the other 3 corners are ...
168                   SE = SW + (length, 0)   # ... relative to NW
169                   NE = NW + (length, 0)   #
170
171                   N = [NW, NE]
172                   S = [SW, SE]
173                   W = [NW, SW]
174                   E = [NE, SE]
175           }
176   }
177
178   boolean doorHitTable, cableIsShort
179   doorHitTable = includes(circle(door/hinge, door/width),table/NW)
180   cableIsShort = dist(cable.1, cable.2) > cablelength
181
182   ?prohibit_touch = OFF;
183
184   /*
185    * End of the original DoNaLD script for specifying a room
186    */
187
188   %scout
189
190   /*
191    * Scout description for the layout of the views of the room and buttons
192    */
193
194   display scr, basicScreen;
195   window don1, don2;
196   window monDoor, monCable;
197   point monDoorPos, monCablePos;
198   string monDoorStr, monCableStr;
199   integer _doorHitTable, _cableIsShort, _door_open;
200   integer _plug, _plug1;
201   point p1, q1, p2, q2;
202   integer zoomSize;
203   point zoomPos;
204   point tblMenuRef, miscMenuRef, zoomMenuRef;
205   point plugButtonPos, doorButtonPos;
206   string plugMenu, doorMenu;
207   window plugButton, doorButton;
208   point tblMenuHeaderPos, tblUpPos, tblDownPos, tblLeftPos, tblRightPos;
209   string tblMenuHeader, tblUpMenu, tblDownMenu, tblLeftMenu, tblRightMenu;
210   window tblHeader, tblUp, tblDown, tblLeft, tblRight;
211   point zoomMenuHeaderPos, zoomUpPos, zoomDownPos, zoomLeftPos, zoomRightPos;
212   string zoomMenuHeader, zoomUpMenu, zoomDownMenu, zoomLeftMenu, zoomRightMenu;
213   window zoomHeader, zoomUp, zoomDown, zoomLeft, zoomRight;
```



```
214
215   p1 = (25, 100);
216   q1 = (225, 300);
217   don1 = {
218           box:    [p1, q1],
219           pict:   "view",
220           type:   DONALD,
221           border: 1
222           sensitive: ON
223   };
224   zoomPos = (500, 500);
225   zoomSize = 500;
226   p2 = (275, 100);
227   q2 = (475, 300);
228   don2 = {
229           box:    [p2, q2],
230           pict:   "view",
231           type:   DONALD,
232           xmin:   zoomPos.1 - zoomSize/2,
233           ymin:   zoomPos.2 - zoomSize/2,
234           xmax:   zoomPos.1 + zoomSize/2,
235           ymax:   zoomPos.2 + zoomSize/2,
236           border: 1
237           sensitive: ON
238   };
239
240   monDoor = {
241           frame:  ([monDoorPos, 1, strlen(monDoorStr)]),
242           string: monDoorStr
243   };
244   monDoorStr = if _doorHitTable then "Table obstructs door" else "" endif;
245   monDoorPos = (25, 50);
246
247   monCable = {
248           frame:  ([monCablePos, 1, strlen(monCableStr)]),
249           string: monCableStr
250   };
251   monCableStr = if _cableIsShort then "Cable is not long enough" else "" endif;
252   monCablePos = (25, 70);
253
254   tblMenuRef = (100, 400);
255   miscMenuRef = (250, 400);
256   zoomMenuRef = (400, 400);
257
258   plugButtonPos = miscMenuRef - (strlen(plugMenu).c / 2, 1.r);
259   plugButton = {
260           frame:  ([plugButtonPos, 1, strlen(plugMenu)]),
261           string: plugMenu,
262           border: 1
263           sensitive: ON
264   };
265   plugMenu = if _plug == _plug1 then "9:Use Plug 2" else "9:Use Plug 1" endif;
266
267   doorButtonPos = miscMenuRef + (-strlen(doorMenu).c / 2, 1.r);
268   doorButton = {
269           frame:  ([doorButtonPos, 1, strlen(doorMenu)]),
270           string: doorMenu,
271           border: 1
272           sensitive: ON
273   };
274   doorMenu = if _door_open then "10:Close Door" else "10:Open Door" endif;
275
276   tblMenuHeader = "Table Position";
277   tblMenuHeaderPos = tblMenuRef - ((strlen(tblMenuHeader)/2).c, 4.r);
278   tblHeader = {
279           frame:  ([tblMenuHeaderPos, 1, strlen(tblMenuHeader)]),
280           string: tblMenuHeader
281   };
282
283   tblUpPos = tblMenuRef - ((strlen(tblUpMenu)/2).c, 2.r);
284   tblUp = {
```

```
285             frame:  ([tblUpPos, 1, strlen(tblUpMenu)]),
286             string: tblUpMenu,
287             border: 1
288             sensitive: ON
289     };
290     tblUpMenu = "1:Up";
291
292     tblDownPos = tblMenuRef + (-(strlen(tblDownMenu)/2).c, 2.r);
293     tblDown = {
294             frame:  ([tblDownPos, 1, strlen(tblDownMenu)]),
295             string: tblDownMenu,
296             border: 1
297             sensitive: ON
298     };
299     tblDownMenu = "2:Down";
300
301     tblLeftPos = tblMenuRef - ((strlen(tblLeftMenu) + 1).c, 0);
302     tblLeft = {
303             frame:  ([tblLeftPos, 1, strlen(tblLeftMenu)]),
304             string: tblLeftMenu,
305             border: 1
306             sensitive: ON
307     };
308     tblLeftMenu = "3:Left";
309
310     tblRightPos = tblMenuRef + (1.c, 0);
311     tblRight = {
312             frame:  ([tblRightPos, 1, strlen(tblRightMenu)]),
313             string: tblRightMenu,
314             border: 1
315             sensitive: ON
316     };
317     tblRightMenu = "4:Right";
318
319     zoomMenuHeader = "Zoom Position";
320     zoomMenuHeaderPos = zoomMenuRef - ((strlen(zoomMenuHeader)/2).c, 4.r);
321     zoomHeader = {
322             frame:  ([zoomMenuHeaderPos, 1, strlen(zoomMenuHeader)]),
323             string: zoomMenuHeader
324     };
325
326     zoomUpPos = zoomMenuRef - ((strlen(zoomUpMenu)/2).c, 2.r);
327     zoomUp = {
328             frame:  ([zoomUpPos, 1, strlen(zoomUpMenu)]),
329             string: zoomUpMenu,
330             border: 1
331             sensitive: ON
332     };
333     zoomUpMenu = "5:Up";
334
335     zoomDownPos = zoomMenuRef + (-(strlen(zoomDownMenu)/2).c, 2.r);
336     zoomDown = {
337             frame:  ([zoomDownPos, 1, strlen(zoomDownMenu)]),
338             string: zoomDownMenu,
339             border: 1
340             sensitive: ON
341     };
342     zoomDownMenu = "6:Down";
343
344     zoomLeftPos = zoomMenuRef - ((strlen(zoomLeftMenu) + 1).c, 0);
345     zoomLeft = {
346             frame:  ([zoomLeftPos, 1, strlen(zoomLeftMenu)]),
347             string: zoomLeftMenu,
348             border: 1
349             sensitive: ON
350     };
351     zoomLeftMenu = "7:Left";
352
353     zoomRightPos = zoomMenuRef + (1.c, 0);
354     zoomRight = {
355             frame:  ([zoomRightPos, 1, strlen(zoomRightMenu)]),
```

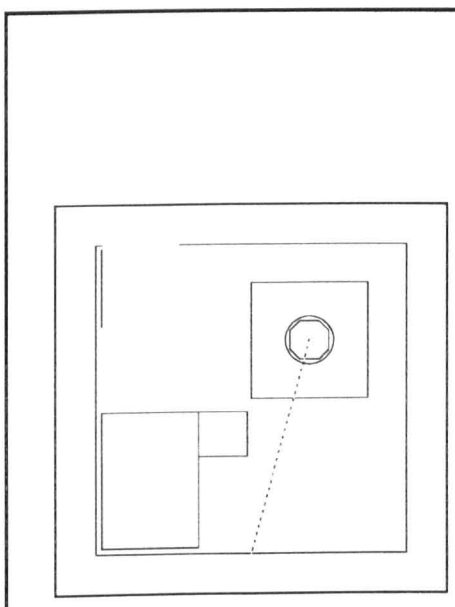


```
356             string: zoomRightMenu,
357             border: 1
358             sensitive: ON
359     };
360     zoomRightMenu = "8:Right";
361
362     basicScreen = < tblHeader / tblUp / tblDown / tblLeft / tblRight /
363                     zoomHeader / zoomUp / zoomDown / zoomLeft / zoomRight /
364                     plugButton / doorButton /
365                     don1 / don2 >;
366
367     scr = if _doorHitTable then
368             append(basicScreen, 1, monDoor)
369           else
370             basicScreen
371           endif;
372
373     screen = if _cableIsShort then
374                     append(scr, 1, monCable)
375              else
376                     scr
377              endif;
378
379     %eden
380
381     /*
382      *      action to effect state change
383      */
384
385     proc user_input : input
386     {
387             switch (input) {
388             case 1: _table_SW = vector_add(_table_SW, cart(0, 100));        break;
389             case 2: _table_SW = vector_sub(_table_SW, cart(0, 100));        break;
390             case 3: _table_SW = vector_sub(_table_SW, cart(100, 0));        break;
391             case 4: _table_SW = vector_add(_table_SW, cart(100, 0));        break;
392             case 5: zoomPos = pt_add(zoomPos, [0, 100]);                    break;
393             case 6: zoomPos = pt_subtract(zoomPos, [0, 100]);               break;
394             case 7: zoomPos = pt_subtract(zoomPos, [100, 0]);               break;
395             case 8: zoomPos = pt_add(zoomPos, [100, 0]);                    break;
396             case 9: if (_plug == _plug1) {
397                             _plug is _plug2;
398                     } else {
399                             _plug is _plug1;
400                     }
401                     break;
402             case 10: _door_open = !_door_open;                              break;
403             }
404     }
405
406     /*
407      *      actions for interpreting mouse actions
408      */
409
410     proc plugButton_to_input : plugButton_mouse_1 {
411             if (plugButton_mouse_1[2] == 4) input = 9;
412     }
413
414     proc doorButton_to_input : doorButton_mouse_1 {
415             if (doorButton_mouse_1[2] == 4) input = 10;
416     }
417
418     proc tblUp_to_input : tblUp_mouse_1 {
419             if (tblUp_mouse_1[2] == 4) input = 1;
420     }
421
422     proc tblDown_to_input : tblDown_mouse_1 {
423             if (tblDown_mouse_1[2] == 4) input = 2;
424     }
425
426     proc tblLeft_to_input : tblLeft_mouse_1 {
```

```
427             if (tblLeft_mouse_1[2] == 4) input = 3;
428     }
429
430     proc tblRight_to_input : tblRight_mouse_1 {
431             if (tblRight_mouse_1[2] == 4) input = 4;
432     }
433
434     proc zoomUp_to_input : zoomUp_mouse_1 {
435             if (zoomUp_mouse_1[2] == 4) input = 5;
436     }
437
438     proc zoomDown_to_input : zoomDown_mouse_1 {
439             if (zoomDown_mouse_1[2] == 4) input = 6;
440     }
441
442     proc zoomLeft_to_input : zoomLeft_mouse_1 {
443             if (zoomLeft_mouse_1[2] == 4) input = 7;
444     }
445
446     proc zoomRight_to_input : zoomRight_mouse_1 {
447             if (zoomRight_mouse_1[2] == 4) input = 8;
448     }
449
450     proc don1_to_tableSW : don1_mouse {
451         auto mx, my;
452         mx = don1_mouse[4];
453         my = don1_mouse[5];
454         if (don1_mouse[2] == 4) {
455             if (_table_SW[2] < mx && mx < _table_NE[2] &&
456                 _table_SW[3] < my && my < _table_NE[3]) move_table = 1;
457             old_table_SW = _table_SW;
458             old_mouse_pos = [mx, my];
459         }
460         if (don1_mouse[2] == 5) {
461             if (move_table == 1) {
462                 _table_SW = cart(old_table_SW[2] + mx - old_mouse_pos[1],
463                                  old_table_SW[3] + my - old_mouse_pos[2]);
464                 move_table = 0;
465             }
466         }
467     }
468
469     proc don2_to_tableSW : don2_mouse {
470         auto mx, my, xmin, xmax, ymin, ymax, m, c, cpi;
471         if (don2_mouse[2] == 4) {
472             mx = don2_mouse[4];
473             my = don2_mouse[5];
474             xmin = dotint(don2, 6);
475             ymin = dotint(don2, 7);
476             xmax = dotint(don2, 8);
477             ymax = dotint(don2, 9);
478             m = (ymax - ymin) / (xmax - xmin);
479             c = ymin - m * xmin;
480             cpi = ymax + m * xmin;
481             if ((my - m * mx) > c)
482                     input = ((my + m * mx) > cpi) ? 5 : 7;
483             else
484                     input = ((my + m * mx) > cpi) ? 8 : 6;
485         }
486     }
487
488     %scout
```

The Room Example

F.2. Sample Output



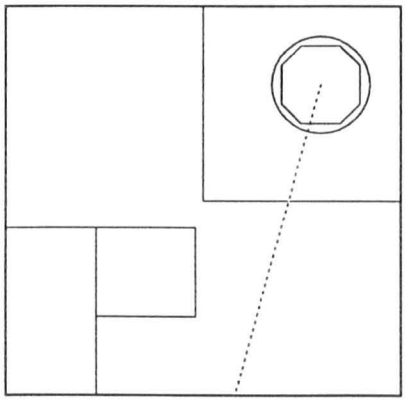


Table Position

1:Up

3:Left 4:Right

2:Down

9:Use Plug 2

10:Close Door

Zoom Position

5:Up

7:Left 8:Right

6:Down

Appendix G

The Vehicle Cruise Control Simulation Example

G.1. The LSD Specification of the Simulation

G.2. EDEN Implementation of the LSD Specification

G.3. Scout Graphical Interface of the Simulation

G.4. Sample Output

The Vehicle Cruise Control Simulation Example

G.1. The LSD Specification of the Simulation

```
/*****************************************************************
  file   : cruise.lsd
  date   : 22.08.91
  author : I.Bridge
  notes  : description of agents for a 'cruise control' system
           using a version of LSD developed by the author
*****************************************************************/


TYPEDEF
  pushBtn_Type      = ENUM (pbUp, pbDown)
  cruiseStts_Type   = ENUM (csOn, csMaintain, csOff)
  throttleStts_Type = ENUM (tsOff, tsMan, tsAuto)
  engineStts_Type   = ENUM (esOn, esOff)

  accelPos_Type     = REAL(0.0, 1.0)            /* normalised position */
  brakePos_Type     = REAL(0.0, 1.0)            /* normalised position */
  throttlePos_Type  = REAL(0.0, 1.0)            /* normalised position */
  gradient_Type     = REAL(-100.0, 100.0)       /* percent             */


AGENT control_panel (
CONST
  minCruiseSpeed = 30      /* [km h^-1] */
  maxCruiseSpeed = 100     /* [km h^-1] */
STATE
  cruiseStts  : cruiseStts_Type = csOff
  cruiseSpeed : INT             = minCruiseSpeed
  onBtn       : pushBtn_Type    = pbUp /* switch on cruise controller      */
  offBtn      : pushBtn_Type    = pbUp /* switch off cruise controller     */
  incrBtn     : pushBtn_Type    = pbUp /* increment cruise speed           */
  decrBtn     : pushBtn_Type    = pbUp /* decrement cruise speed           */
  setBtn      : pushBtn_Type    = pbUp /* set cruise speed = current speed */
  resBtn      : pushBtn_Type    = pbUp /* resume cruise speed              */
  manBtn      : pushBtn_Type    = pbUp /* revert to manual operation       */
INTERFACE
  onBtn       : IN
  offBtn      : IN
  incrBtn     : IN
  decrBtn     : IN
  setBtn      : IN
  resBtn      : IN
  brakePos    : IN
  speed       : IN
  engineStts  : IN
  cruiseStts  : OUT
  cruiseSpeed : OUT
DERIVATE
  braking is (brakePos != 0)
  press_btn(pushBtnSt) is ((pushBtnSt' == pbUp) && (pushBtnSt == pbDown))
PROTOCOL
  (press_btn(onBtn) && (engineStts == esOn))
    --> cruiseStts = csOn
  (press_btn(offBtn) || (engineStts == esOff))
    --> cruiseStts = csOff
  ((press_btn(incrBtn) && (cruiseStts != csOff) && (cruiseSpeed < maxCruiseSpeed))
    --> cruiseSpeed++
  ((press_btn(decrBtn) && (cruiseStts != csOff) && (cruiseSpeed > minCruiseSpeed))
    --> cruiseSpeed--
  ((press_btn(setBtn) && (cruiseStts == csOn))
    --> cruiseSpeed = speed; cruiseStts = csMaintain;
  ((press_btn(resBtn) && (cruiseStts == csOn))
    --> cruiseStts = csMaintain
  ((braking  || pressBtn(manBtn)) && (cruiseStts == csMaintain))
    --> cruiseStts = csOn
)


AGENT throttle_manager (
CONST
  GainK = 0.5  /* auto throttle controller gain          */
  TimeK = 2.0  /* auto throttle controller time constant */
STATE
  throttleStts      : throttleStts_Type
  throttlePos       : throttlePos_Type
  deltaAutoThrottle : REAL
INTERFACE
  speed         : IN
  cruiseSpeed   : IN
  cruiseStts    : IN
  engineStts    : IN
  accelPos      : IN
  throttlePos   : OUT
DERIVATE
  speedErr is cruiseSpeed - measSpeed
  deltaAutoThrottle is ((GainK * speedErr) - throttlePos') / TimeK
  throttlePos is
    (throttleStts == tsOff)  ? 0.0 :
    (throttleStts == tsMan)  ? accelPos :
    (throttleStts == tsAuto) ? integ_wrt_time(deltaAutoThrottle, accelPos)
PROTOCOL
  (engineStts == esOff)
    --> throttleStts = tsOff
  ((cruiseStts != csMaintain) && (engineStts == esOn))
    --> throttleStts = tsMan
  ((cruiseStts == csMaintain) && (engineStts == esOn))
    --> throttleStts = tsAuto
)


AGENT engine (
CONST
  maxEngineTorque = 74500
STATE
  engineStts   : engineStts_Type = esOff
INTERFACE
  engineStts   : IN OUT
  engineTorque  : OUT
  throttlePos  : IN
DERIVATE
  engineTorque is
    (engineStts == esOn)  ? maxEngineTorque * throttlePos :
    (engineStts == esOff) ? 0;
)


AGENT vehicle_dynamics (
CONST
  mass   = 2500    /* total mass of car & contents [kg]            */
  windK  = 50.0    /* wind resistance factor [N m^2 s^2]           */
  rollK  = 10.0    /* rolling resistance factor [N m^-1 s]         */
  gravK  = 9.81    /* acceleration due to gravity [m s^-2]         */
  brakK  = 1500.0  /* viscous friction braking constant [N m^-1 s] */
  forcK  = 40.0    /* torque to force conversion                   */
  sticK  = 100.0   /* static friction                              */
STATE
  actSpeed : REAL := 0.0  /* actual speed             */
  accel    : REAL
  windF    : REAL         /* wind resistance force    */
  rollF    : REAL         /* rolling resistance force */
  gradF    : REAL         /* gradient force           */
  tracF    : REAL         /* engine traction force    */
INTERFACE
  gradient     : IN
  speed        : OUT
  engineTorque : IN
  brakePos     : IN
DERIVATE
  windF is windK * pwr(actSpeed',2)
  rollF is rollK * actSpeed'
  gradF is gravK * mass * sin(gradient * pi / 200)
  brakF is brakK * actSpeed' * brakePos
```

```
143     tracF is forcK * engineTorque;
144     sticF is sticK * sgn(actSpeed') * bound(actSpeed', -0.01, 0.01)
145     accel is (tracF - brakF - gradF - rollF - windF - sticF) / mass
146     actSpeed is integ_wrt_time(accel, 0)
147   )
148
149
150   AGENT speed_transducer (
151   CONST
152     wheelDiam   = 0.45           /* wheel diameter [m]           */
153     wheelCirc   = pi * wheelDiam /* wheel circumference [m]      */
154     wheelPuls   = 8;             /* pulses per wheel revolution  */
155     countPeriod = 0.2            /* counter/timer period [s]     */
156     maxCountVal = 65535          /* assumes 16-bit counter       */
157   STATE
158     measSpeed : REAL
159     pulseRate : REAL             /* wheel revs/sec [s^-1]       */
160     countVal  : REAL             /* timer/counter value [s^-1]  */
161   INTERFACE
162     actSpeed  : IN
163     measSpeed : OUT
164   DERIVATE
165     pulseRate is int(actSpeed * wheelPuls / wheelCirc)
166     countVal is int(pulseRate * countPeriod) % maxCountVal
167     measSpeed is (countVal * wheelCirc) / countPeriod
168   )
169
170
171   AGENT driver (
172   STATE
173     accelPos : accelPos_Type
174     brakePos : brakePos_Type
175   INTERFACE
176     engineStts  : IN OUT
177     cruiseSpeed : IN
178     accelPos    : OUT
179     brakePos    : OUT
180     onBtn       : OUT
181     offBtn      : OUT
182     incrBtn     : OUT
183     decrBtn     : OUT
184     setBtn      : OUT
185     resBtn      : OUT
186   /* comment required */
187   )
188
189
190   AGENT environment (
191   STATE
192     gradient : gradient_Type  /* gradient (%) */
193   INTERFACE
194     gradient : OUT
195   /* comment required */
196   )
197
```

The Vehicle Cruise Control Simulation Example

G.2. EDEN Implementation of the LSD Specification

```
1   /*****************************************************************************
2     file      : main.e
3     date      : 04.09.91
4     author    : I.Bridge
5     notes     : implementation of 'cruise.lsd'
6   *****************************************************************************/
7
8   include("io.e");
9   include("utils.e");
10  include("enum.e");
11  include("macros.e");
12  include("control_panel.e");
13  include("throttle_manager.e");
14  include("engine.e");
15  include("vehicle_dynamics.e");
16  include("speed_transducer.e");
17  include("driver.e");
18  include("environment.e");
19
20  iPeriod = 0.01;                     /* [s] */
21
22  speed is round(mps_to_mph(measSpeed),2);  /* [mph] */
23  update is 100;                            /* clock ticks per report */
24  report = 0;
25
26  proc _report  : iClock {
27    /* display time, speed and throttle update rate */
28    if ((iClock % update) == 0) {
29      sampleClk = iClock;
30      _curSpeed = speed;
31      sampleHDisp = HDisplacement;
32      sampleVDisp = VDisplacement;
33      _sampleThrottlePos = throttlePos;
34      _gradient = gradient * pi / 200.0;
35      _windF = windF;
36      _gravF = gravK * mass;
37      _tracF = tracF;
38      _brakF = brakF;
39  if (report) {
40      write("t=", float(iClock*iPeriod));
41      write("\t speed[mph]=", speed);
42      write("\t throttlePos=", throttlePos);
43      writeln("\t");
44      writeln("\tdistance[meter] = ", distance, "\tgradient = ", gradient);
45      writeln("\tHD[meter] = ", HDisplacement, "\tVD[meter] = ", VDisplacement);
46      writeln("\twind = ", windF, "\tgravity = ", gravF, "\ttrac = ", tracF);
47  }
48    }
49  }
50
51  proc init {
52    /* initialise system */
53    iClock = 0;
54    init_control_panel();
55    init_throttle_manager();
56    init_engine();
57    init_vehicle_dynamics();
58    init_driver();
59    init_environment();
60  }
61
62  break1 = 1000;
63  func go {
64    /* start clock */
65    iClock++;
66    while ((iClock % break1) != 0) { iClock++; }
67    /* for (;;) { iClock++; } */
68  }
69
70  init();
71
```



```
72  /* travelling instruction to the car! */
73  func dist_to_grad {
74          para dist;
75          return sin(int(dist) % 1000 / 1000.0 * 2 * pi) * 6.3;
76  }
77  gradient is dist_to_grad(distance);
78  gradient_in_rad is gradient / 200.0 * pi;
79  integ_wrt_time("actSpeed", "distance", "0.0");
80  integ_wrt_time("actSpeed * cos(gradient_in_rad)", "HDisplacement", "0.0");
81  integ_wrt_time("actSpeed * sin(gradient_in_rad)", "VDisplacement", "0.0");
```

```
1   /*****************************************************************************
2     file      : io.e
3     date      : 06.09.91
4     author    : I.Bridge
5     notes     : input/output functions used by implementation of 'cruise.lsd'
6   *****************************************************************************/
7
8   func get_enum {
9     /* output current value of 'enumVal' as a string and read it's value as a
10       string (value unchanged if only <CR> entered) and return it's enumerated
11       value */
12    para enumVal, enumStrS;
13    auto selValid, selStr;
14    selValid = FALSE;
15    while (!selValid) {
16      write(enumStrS[enumVal], " : ");
17      selStr = "";
18      gets(selStr);
19      if (selStr == "") {
20        selEnum = enumVal;
21        selValid = TRUE;
22      } else {
23        if (member(selStr, enumStrS)) {
24          selEnum = nth(selStr, enumStrS);
25          selValid = TRUE;
26        }
27      }
28    }
29    return selEnum;
30  }
31
32  func get_real {
33    /* output current value of 'realVal' and read it's new value as a string
34       (value unchanged if only <CR> entered) and return it's value as a real */
35    para realVal;
36    auto entryValid, entryStr, entryVal;
37    entryValid = FALSE;
38    while (!entryValid) {
39      write(realVal, " : ");
40      entryStr = "";
41      gets(entryStr);
42      if (entryStr == "") {
43        entryVal = realVal;
44        entryValid = TRUE;
45      } else {
46        if (valid_real_string(entryStr)) {
47          entryVal = float(entryStr);
48          entryValid = TRUE;
49        }
50      }
51    }
52    return entryVal;
53  }
54
55  func valid_real_string {
56    /* true if string is real number */
57    para realStr;
58    auto i, validChars;
59    validChars = "0123456789.-+";
60    for (i = 1; i <= realStr#; i++) {
61      if (!member(realStr[i], validChars)) {
62        return FALSE;
63      }
64    }
65    return TRUE;
66  }
67
```

```
1     /*****************************************************************************
2       file      : utils.e
3       date      : 06.09.91
4       author    : I.Bridge
5       notes     : utility functions used by implementation of 'cruise.lsd'
6     *****************************************************************************/
7
8     func member {
9       /* TRUE if 'item' is a member of 'itemList' */
10      para item, itemList;
11      auto i;
12      for (i = 1; i <= itemList#; i++) {
13        if (item == itemList[i]) {
14          return TRUE;
15        }
16      }
17      return FALSE;
18    }
19
20    func nth {
21      /* returns position of 'item' in 'itemList' */
22      para item, itemList;
23      auto i;
24      for (i = 1; i <= itemList#; i++) {
25        if (item == itemList[i]) {
26          return i;
27        }
28      }
29    }
30
31    func sgn {
32      /* returns +1 or -1 if 'val' is +ve or -ve respectively */
33      para val;
34      if (val >= 0) {
35        return 1;
36      } else {
37        return -1;
38      }
39    }
40
41    func bound {
42      /* returns +1 if 'val' lies within range 'lowVal' - 'uppVal' */
43      para val, lowVal, uppVal;
44      if ((val >= lowVal) && (val <= uppVal)) {
45        return 1.0;
46      } else {
47        return 0.0;
48      }
49    }
50
51    func limit {
52      para val, lowVal, uppVal;
53      if (val < lowVal) { return lowVal; }
54      if (val > uppVal) { return uppVal; }
55      return val;
56    }
57
58    func round {
59      /* return 'val' rounded to 'nDec' decimal places */
60      para val, nDec;
61      auto tmp;
62      tmp = val * pow(10.0, float(nDec)) + 0.5;
63      return int(tmp) / pow(10.0, float(nDec));
64    }
65
66    func mph_to_mps {
67      /* convert miles/hour to metres/sec */
68      para mph;
69      return 0.448 * mph;
70    }
71
```



```
72    func mps_to_mph {
73      /* convert metres/sec to miles/hour */
74      para mps;
75      return 2.232 * mps;
76    }
77
```

```
1   /*****************************************************************************
2     file      : enum.e
3     date      : 04.09.91
4     author    : I.Bridge
5     notes     : enumerated types for implementation of 'cruise.lsd'
6   *****************************************************************************/
7
8   /*
9     ENUM boolean
10  */
11
12    FALSE = 0;
13    TRUE  = 1;
14
15  /*
16    ENUM pushBtn_Type
17  */
18
19    pbUp   = 1;
20    pbDown = 2;
21
22  /*
23    ENUM cruiseStts_Type
24  */
25
26    csOn       = 1;
27    csMaintain = 2;
28    csOff      = 3;
29
30  /*
31    ENUM throttleStts_Type
32  */
33
34    tsOff  = 1;
35    tsMan  = 2;
36    tsAuto = 3;
37
38  /*
39    ENUM engineStts_Type
40  */
41
42    esOn  = 1;
43    esOff = 2;
44
45  /*
46    string representations of enumerated values
47  */
48
49    pushBtn_EnumStr      = ["up","down"];
50    cruiseStts_EnumStr   = ["on","maintain","off"];
51    throttleStts_EnumStr = ["off","manual","automatic"];
52    engineStts_EnumStr   = ["on","off"];
53
```

```
/*****************************************************************************
  file      : macros.e
  date      : 04.09.91
  author    : I.Bridge
  notes     : general purpose macro generation functions used in implementation
              of 'cruise.lsd'
*****************************************************************************/


/*---------------------------------------------------------------------------
function macro(macro_str, para_str1, para_str2, .., para_strN)
  Expands 'macro_str' by substituting 'para_strI' for "?I" and returns the
  resultant string (ref.: Edward Yung, M.Sc. thesis, '89, vol.2).
---------------------------------------------------------------------------*/

func macro {
  auto i, j, l, m, n, c, s;
  s = "";
  l = (m = $1)#;
  shift;
  i = 1;
  while (i <= l) {
    for (j = i; j <= l && m[j] != '?'; j++);
    if (i != j) s = s // substr(m, i, j - 1);
    if (j <= l) {
      j++;
      n = (c = (j > l) ? '?' : m[j]) - '0';
      s = s // ((1 <= n && n <= $#) ? $[n] : c);
    }
    i = j + 1;
  }
  return s;
}

/*---------------------------------------------------------------------------
procedure integ_wrt_time(inp_str, out_str, init_str)
  Defines a trapezoidal integrator of the form :-
    out[n] = ((in[n-1] + in[n]) * iPeriod/2) + out[n-1]
  where 'out' is initialised to 'init' and 'iPeriod' is the integration
  period.
  A transition of the variable 'iClock' is the trigger used to invoke an
  integration cycle. The variable 'strcat(inp_str, "iVal")' is generated as an
  accumulator for the implementation of the discrete integrator.
  (N.B. a bug/feature in 'execute' causes the body of a 'proc' defined
  within a 'macro' to be executed rather than simply defined.)
---------------------------------------------------------------------------*/

/*
proc integ_wrt_time {
  para inp_str, out_str, init_str;
  execute(macro(
    "
    proc integ_?1 : iClock {
      ?2 = ?1_iVal + (?1 * iPeriod / 2);
      ?1_iVal = ?2 + (?1 * iPeriod / 2);
    }
    ",
    inp_str, out_str
  ));
  execute(macro(
    "
    ?1 = ?2;
    ",
    out_str, init_str
  ));
}
*/

proc integ_wrt_time {
  para inp_str, out_str, init_str;
  execute(macro(
```



```
    "
    ?1_iVal = ?2;
    ?1 = ?2;
    ",
    out_str, init_str
  ));
  execute(macro(
    "
    proc integ_?2 : iClock {
      ?2 = ?2_iVal + (?1 * iPeriod / 2);
      ?2_iVal = ?2 + (?1 * iPeriod / 2);
    }
    ",
    inp_str, out_str
  ));
}
```

```
/*****************************************************************************
  file      : control_panel.e
  date      : 22.08.91
  author    : I.Bridge
  notes     : implementation of 'control_panel' agent of 'cruise.lsd'
  interface : onBtn       : IN
              offBtn      : IN
              incrBtn     : IN
              decrBtn     : IN
              setBtn      : IN
              resBtn      : IN
              brakePos    : IN
              speed       : IN
              engineStts  : IN
              cruiseStts  : OUT
              cruiseSpeed : OUT
*****************************************************************************/
/*
  constants :-
*/

  minCruiseSpeed_mph = 20.0;          /* [miles/hour] */
  maxCruiseSpeed_mph = 70.0;          /* [miles/hour] */

/*
  initialisations :-
*/

proc init_control_panel (
  onBtn = pbUp;
  offBtn = pbDown;
  onBtn_prev = pbUp;
  offBtn_prev = pbUp;
  incrBtn = pbUp;
  decrBtn = pbUp;
  setBtn = pbUp;
  resBtn = pbUp;
  manBtn = pbUp;
  cruiseStts = csOff;
  cruiseSpeed_mph = minCruiseSpeed_mph;
)

init_control_panel();

/*
  derivate :-
*/

cruiseSpeed is mph_to_mps(cruiseSpeed_mph);
braking is (brakePos != 0.0);

/*
  protocol :-
*/

proc agent_control_1 :onBtn (         /* on */
  if ((onBtn == pbDown) && (onBtn_prev == pbUp)) (
    if (engineStts == esOn) (
      offBtn = pbUp;
      cruiseStts = csOn;
    ) else
      onBtn = pbUp;
  )
  onBtn_prev = onBtn;
)

proc agent_control_2 : offBtn (       /* off */
  if ((offBtn == pbDown) && (offBtn_prev == pbUp)) (
    cruiseStts = csOff;
    onBtn = pbUp;
  )
  offBtn_prev = offBtn;
)

proc agent_control_2b : engineStts ( /* off */
  if (engineStts == esOff) (
    offBtn = pbDown;
  )
)

/*
proc agent_control_3 : incrBtn (      /* increment */
  if ((incrBtn == pbDown) && (incrBtn_prev == pbUp) &&
      (cruiseStts != csOff) && (cruiseSpeed_mph < maxCruiseSpeed_mph)) (
        cruiseSpeed_mph++;
  )
  incrBtn_prev = incrBtn;
)

proc agent_control_4 : decrBtn (      /* decrement */
  if ((decrBtn == pbDown) && (decrBtn_prev == pbUp) &&
      (cruiseStts != csOff) && (cruiseSpeed_mph > minCruiseSpeed_mph)) (
        cruiseSpeed_mph--;
  )
  decrBtn_prev = decrBtn;
)
*/

proc agent_control_5 : setBtn (       /* set */
  if ((setBtn == pbDown) && (setBtn_prev == pbUp) &&
      (cruiseStts == csOn)) (
    cruiseSpeed_mph = speed;
    cruiseStts = csMaintain;
  )
  setBtn_prev = setBtn;
)

proc agent_control_6 : resBtn (       /* resume */
  if ((resBtn == pbDown) && (resBtn_prev == pbUp) &&
      (cruiseStts == csOn)) (
    cruiseStts = csMaintain;
  )
  resBtn_prev = resBtn;
)

proc agent_control_7a : braking (     /* brake */
  if ((braking) && (cruiseStts == csMaintain))  (
    cruiseStts = csOn;
  )
)

proc agent_control_7b : manBtn (    /* manual */
  if ((manBtn == pbDown) && (manBtn_prev == pbUp) &&
      (cruiseStts == csMaintain))  (
        cruiseStts = csOn;
  )
  manBtn_prev = manBtn;
)
```

```
1   /*****************************************************************************
2     file      : throttle_manager.e
3     date      : 04.09.91
4     author    : I.Bridge
5     notes     : implementation of 'throttle_manager' agent of 'cruise.lsd'
6     interface : speed          : IN
7                 cruiseSpeed    : IN
8                 cruiseStts     : IN
9                 engineStts     : IN
10                accelPos       : IN
11                throttlePos    : OUT
12  *****************************************************************************/
13
14  /*
15    constants :-
16  */
17
18    GainK = 100.0; /* auto throttle controller gain          */
19    TimeK = 0.01;  /* auto throttle controller time constant */
20
21  /*
22    initialisations :-
23  */
24
25  func init_throttle_manager {
26    deltaAutoThrottle_iVal = 0.0;   /* NEED TO FIX INTEGTRATER */
27  }
28
29  init_throttle_manager();
30
31  /*
32    derivate
33  */
34
35    speedErr is (cruiseSpeed - measSpeed) / cruiseSpeed;
36
37    /* NB if 'speedErr' is normalised and 'GainK' is 1 then this will */
38    /* normalise the output of the controller                         */
39
40    deltaAutoThrottle is ((GainK * speedErr) - autoThrottle) / TimeK;
41
42    throttlePos is
43      (throttleStts == tsOff)  ?    0.0 :
44      (throttleStts == tsMan)  ?    accelPos :
45   /* (throttleStts == tsAuto) ? */ limit(autoThrottle, accelPos, 1.0);
46
47    proc reset_integrater : throttleStts {
48      /* reset auto throttle whenever 'throttleStts' 'tsAuto' is activated  */
49      if (throttleStts == tsAuto) {
50        integ_wrt_time("deltaAutoThrottle","autoThrottle","accelPos");
51      }
52    }
53
54  /*
55    protocol
56  */
57
58  proc agent_throttle_1 : engineStts {
59    if (engineStts == esOff) {
60      throttleStts = tsOff;
61    }
62  }
63
64  proc agent_throttle_2 : cruiseStts, engineStts {
65    if ((engineStts == esOn) && (cruiseStts != csMaintain)) {
66      throttleStts = tsMan;
67    }
68  }
69
70  proc agent_throttle_3 : cruiseStts, engineStts {
71    if ((engineStts == esOn) && (cruiseStts == csMaintain)) {
```



```
72      throttleStts = tsAuto;
73    }
74  }
75
```

```
1   /*****************************************************************************
2     file      : engine.e
3     date      : 04.09.91
4     author    : I.Bridge
5     notes     : implementation of 'engine' agent of 'cruise.lsd'
6     interface : engineStts   : IN OUT
7                 engineTorque : OUT
8                 throttlePos  : IN
9   *****************************************************************************/
10
11  /*
12    constants :-
13  */
14
15    maxEngineTorque = 180; /* [kg m] */
16
17  /*
18    initialisations :-
19  */
20
21  func init_engine {
22    engineStts = esOff;
23  }
24
25  init_engine();
26
27  /*
28    derivate :-
29  */
30
31    engineTorque is
32      (engineStts == esOn)  ?    maxEngineTorque * throttlePos :
33   /* (engineStts == esOff) ? */ 0.0 ;
34
```

```
1   /*****************************************************************************
2     file      : vehicle_dynamics.e
3     date      : 04.09.91
4     author    : I.Bridge
5     notes     : implementation of 'vehicle_dynamics' agent of 'cruise.lsd'
6     interface : gradient     : IN
7                 speed        : OUT
8                 tracF        : IN
9                 brakePos     : IN
10                engineTorque : IN
11  *****************************************************************************/
12
13  /*
14    constants :-
15  */
16    pi     = 3.14159;
17    mass   = 2500.0;  /* total mass of car & contents [kg]      */
18    windK  = 5.0;     /* wind resistance factor [N m^2 s^2]     */
19    rollK  = 50.0;    /* rolling resistance factor [N m^-1 s]   */
20    gravK  = 9.81;    /* acceleration due to gravity [m s^-2]   */
21    brakK  = 1500.0;  /* braking (viscous) constant [N m^-1 s]  */
22    forcK  = 40.0;    /* torque to force conversion [m^-1]      */
23    sticK  = 100.0;   /* static friction force [N]              */
24
25  /*
26    initialisations :-
27  */
28
29  proc init_vehicle_dynamics {
30    accel_iVal = 0.0;           /* NEED TO FIX INTEGTRATER */
31    actSpeed = 0.0;
32  }
33
34  init_vehicle_dynamics();
35
36  /*
37    derivate :-
38  */
39
40    integ_wrt_time("accel","actSpeed","0.0");
41
42    windF is ((actSpeed >= 0) ? 1 : -1) * windK * pow(actSpeed, 2.0);
43    rollF is rollK * actSpeed;
44    gravF is gravK * mass * sin(gradient * pi / 200);
45    brakF is brakK * brakePos * actSpeed;
46    tracF is forcK * engineTorque;
47    sticF is sticK * sgn(actSpeed) * bound(actSpeed, -0.01, 0.01);
48    accel is (tracF - brakF - gravF - rollF - windF - sticF) / mass;
```

```
1   /*****************************************************************************
2     file      : speed_transducer.e
3     date      : 04.09.91
4     author    : I.Bridge
5     notes     : implementation of 'speed_transducer' agent of 'cruise.lsd'
6     interface : actSpeed  : IN
7                 measSpeed : OUT
8   *****************************************************************************/
9
10  /*
11    constants :-
12  */
13
14    wheelDiam   = 0.45;           /* wheel diameter [m]           */
15    wheelCirc   = pi * wheelDiam; /* wheel circumference [m]      */
16    wheelPuls   = 8;              /* pulses per wheel revolution  */
17    countPeriod = 1.0;            /* counter/timer period [s]     */
18    maxCountVal = 65535;          /* assumes 16-bit counter       */
19
20  /*
21    derivate :-
22  */
23
24    pulseRate is int(actSpeed * wheelPuls / wheelCirc);
25    countVal is int(pulseRate * countPeriod) % maxCountVal;
26    measSpeed is (countVal * wheelCirc) / (countPeriod * wheelPuls);
27
```

```
/*****************************************************************************
  file      : driver.e
  date      : 04.09.91
  author    : I.Bridge
  notes     : implementation of '????' agent of 'cruise.lsd'
  interface : engineStts  : IN OUT
              cruiseSpeed : IN
              accelPos    : OUT
              brakePos    : OUT
              onBtn       : OUT
              offBtn      : OUT
              incrBtn     : OUT
              decrBtn     : OUT
              setBtn      : OUT
              resBtn      : OUT
*****************************************************************************/

/*
  initialisations :-
*/

func init_driver {
  accelPos = 0.0;
  brakePos = 0.0;
}

init_driver();

/*
  prompt for driver input :-
*/

func get_engine_status {
  write("engine status [on,off] : ");
  engineStts = get_enum(engineStts, engineStts_EnumStr);
}

func get_button_states {
  auto i, btnStrS;
  btnStrS = ["onBtn","offBtn","incrBtn","decrBtn","setBtn","resBtn","manBtn"];
  for (i = 1; i <= btnStrS#; i++) {
    write(btnStrS[i], " [up,down] : ");
    execute(macro(
      "
      ?1 = get_enum(?1, pushBtn_EnumStr);
      ",
      btnStrS[i]
    ));
  }
}

func get_brake_pos {
  write("brake position : ");
  brakePos = get_real(brakePos);
}

func get_accel_pos {
  write("accelerator position : ");
  accelPos = get_real(accelPos);
}

func driver {
  writeln("cruise speed[mph] : ", mps_to_mph(cruiseSpeed));
  get_engine_status();
  get_button_states();
  get_brake_pos();
  get_accel_pos();
}
```

```
1   /*****************************************************************************
2     file      : environment.e
3     date      : 08.09.91
4     author    : I.Bridge
5     notes     : implementation of 'environment' agent of 'cruise.lsd'
6     interface :
7   *****************************************************************************/
8
9   func init_environment {
10    gradient = 0.0;
11  }
12
13  init_environment();
14
15  func environment {
16    write("gradient[%] : ");
17    gradient = get_real(gradient);
18  }
19
```

The Vehicle Cruise Control Simulation Example

G.3. Scout Graphical Interface of the Simulation

```
%eden
include("main.e");
%scout
integer pbUp, pbDown;

# speedometer

window speedometer;
point p, q;
integer maxx, minx, maxy, miny;
speedometer = {
        type: DONALD,
        box: [p,q],
        pict: "SPEEDO",
        border: 1,
        xmin:-250,
        xmax:250,
        ymin:-250,
        ymax:250
};

p = {minx,maxy}; q = {maxx, miny};
integer width;
integer height;
maxx = minx + width;
maxy = miny - height;
minx = 275;
miny = 210;
width = 200;
height = 200;

# cruise speed windows

integer cruiseSpeed_mph;
point crOrg;
window crSpdTitle, crUpWin, crDownWin;
window crSpdLCD, crSetBtn, crResumeBtn, crManualBtn;
integer incrBtn, decrBtn;
crOrg = {10, 60};
crSpdTitle = {
        string: "Cruise Speed",
        frame: ([crOrg, crOrg + {20.c + 10, 1.r}]),
        alignment: CENTRE
};
crUpWin = {
        type: DONALD,
        pict: "UPARROW",
        box: [crOrg + {0, 1.r + 3}, 1, 2],
        bgcolor: if incrBtn == pbUp then "white" else "black" endif,
        fgcolor: if incrBtn == pbUp then "black" else "white" endif,
        border: 1,
        sensitive: ON
};
crDownWin = {
        type: DONALD,
        pict: "UPARROW",
        box: [crOrg + {0, 2.r + 6}, 1, 2],
        xmin: 1000, ymin: 1000, xmax:0, ymax: 0,
        bgcolor: if decrBtn == pbUp then "white" else "black" endif,
        fgcolor: if decrBtn == pbUp then "black" else "white" endif,
        border: 1,
        sensitive: ON
};
point crSpdLCDOrg;
crSpdLCDOrg =  crUpWin.box.nw + {2.c + 3, 0};
crSpdLCD = {
        string: itos(cruiseSpeed_mph),
        frame: ([crSpdLCDOrg, crSpdLCDOrg + {18.c + 9, 1.r + 1}]),
        alignment: CENTRE,
        border: 1
};
```



```
crSetBtn = {
        string: "SET",
        frame: ([crDownWin.box.nw + {2.c + 3, 0}, 1, 6]),
        alignment: CENTRE,
        border: 1,
        sensitive: ON
};
crResumeBtn = {
        string: "RESUME",
        frame: ([crSetBtn.frame.1.nw + {6.c + 4, 0}, 1, 6]),
        alignment: CENTRE,
        border: 1,
        sensitive: ON
};
crManualBtn = {
        string: "MANUAL",
        frame: ([crResumeBtn.frame.1.nw + {6.c + 4, 0}, 1, 6]),
        alignment: CENTRE,
        border: 1,
        sensitive: ON
};
display dispCrSpd;
dispCrSpd = < crSpdTitle / crUpWin / crDownWin / crSpdLCD /
        crSetBtn / crResumeBtn / crManualBtn >;

# cruise controller
point crCntOrg;
window crCntTitle, crOnBtn, crOffBtn, crMtnBtn;
integer cruiseStts, csOn, csOff, csMaintain;
crCntOrg = {30, 140};
crCntTitle = {
        string: "Cruise Status",
        frame: ([crCntOrg, crCntOrg + {16.c + 7, 1.r}]),
        alignment: CENTRE
};
crOnBtn = {
        string: "ON",
        frame: ([crCntOrg + {0, 1.r + 3}, 1, 5]),
        bgcolor: if cruiseStts == csOn || cruiseStts == csMaintain
                then "white" else "black" endif,
        fgcolor: if cruiseStts == csOn || cruiseStts == csMaintain
                then "black" else "white" endif,
        alignment: CENTRE,
        border: 1,
        sensitive: ON
};
crOffBtn = {
        string: "OFF",
        frame: ([crOnBtn.frame.1.nw + {5.c + 3, 0}, 1, 5]),
        bgcolor: if cruiseStts == csOff then "white" else "black" endif,
        fgcolor: if cruiseStts == csOff then "black" else "white" endif,
        alignment: CENTRE,
        border: 1,
        sensitive: ON
};
crMtnBtn = {
        string: "MTN",
        frame: ([crOffBtn.frame.1.nw + {6.c + 3, 0}, 1, 5]),
        bgcolor: if cruiseStts == csMaintain then "white" else "black" endif,
        fgcolor: if cruiseStts == csMaintain then "black" else "white" endif,
        alignment: CENTRE,
        border: 1
};

display dispCrCnt;
dispCrCnt = < crCntTitle / crOnBtn / crOffBtn / crMtnBtn >;

# clock
point clkOrg;
window clkTitle, clkLCD, clkStartBtn, clkStopBtn, clkResetBtn;
integer sampleClk;
```

```
143   clkOrg = (10, 500);
144   clkTitle = {
145           string: "Clock",
146           frame: ([clkOrg, clkOrg + (15.c + 7, 1.r)]),
147           alignment: CENTRE
148   };
149   clkLCD = {
150           string: itos(sampleClk / 100.0),
151           frame: (shift(clkTitle.frame.1, 0, 1.r + 3)),
152           alignment: CENTRE,
153           border: 1
154   };
155   clkStartBtn = {
156           string: "ON",
157           frame: ([clkOrg + (0, 2.r + 6), 1, 5]),
158           alignment: CENTRE,
159           border: 1
160           sensitive: ON
161   };
162   clkStopBtn = {
163           string: "OFF",
164           frame: ([clkStartBtn.frame.1.nw + (5.c + 3, 0), 1, 5]),
165           alignment: CENTRE,
166           border: 1
167           sensitive: ON
168   };
169   clkResetBtn = {
170           string: "RST",
171           frame: ([clkStopBtn.frame.1.nw + (5.c + 3, 0), 1, 5]),
172           alignment: CENTRE,
173           border: 1
174           sensitive: ON
175   };
176
177   display dispClk;
178   dispClk = < clkTitle / clkLCD / clkStartBtn / clkStopBtn / clkResetBtn >;
179
180   # engine on/off
181
182   integer engineStts, esOn;
183   point engineOrg;
184   window engineTitle, ignition;
185   engineOrg = (205, 100);
186   engineTitle = {
187           string: "Engine",
188           frame: ([engineOrg, 1, 6]),
189           alignment: CENTRE
190   };
191   ignition = {
192           string: "\n" // if (engineStts - esOn) then "OFF" else "ON" endif,
193           frame: ([engineTitle.frame.1.s - (3.c, -3), 3, 6]),
194           alignment: CENTRE,
195           border: 1,
196           sensitive: ON
197   };
198
199   display dispEngine;
200   dispEngine = < engineTitle / ignition >;
201
202   # brake & accelerator
203
204   integer BAlength, BAwidth;      # for both brake and accelerator
205   point brakeOrg, accOrg;
206   window brakeTitle, brakePedal, brakeMark1, brakeMark2;
207   window accTitle, accPedal, accMark1, accMark2;
208   BAlength = 200;
209   BAwidth = 50;
210   brakeOrg = (300, 250);
211   accOrg = brakeOrg + (100, 0);
212
213   brakeTitle = {
```



```
214           string: "Brake",
215           frame: ([brakePedal.box.n - (5.c / 2, 2.r), 1, 5]),
216           alignment: CENTRE
217   };
218   brakePedal = {
219           type: DONALD,
220           box: [brakeOrg, brakeOrg + (BAwidth, BAlength)],
221           pict: "BRAKE",
222           xmax: 100,
223           ymax: 100,
224           border: 1,
225           sensitive: ON
226   };
227   brakeMark1 = {
228           string: "100%",
229           frame:  ([brakePedal.box.nw - (5.c, 1.r/2), 1, 4]),
230           alignment: RIGHT
231   };
232   brakeMark2 = {
233           string: "0%",
234           frame:  ([brakePedal.box.sw - (5.c, 1.r/2), 1, 4]),
235           alignment: RIGHT
236   };
237
238   display dispBrake;
239   dispBrake = < brakeTitle / brakePedal / brakeMark1 / brakeMark2 >;
240
241   accTitle = {
242           string: "Accelerator",
243           frame: ([accPedal.box.n - (11.c / 2, 2.r), 1, 11]),
244           alignment: CENTRE
245   };
246   accPedal = {
247           type: DONALD,
248           box: [accOrg, accOrg + (BAwidth, BAlength)],
249           pict: "ACC",
250           xmax: 100,
251           ymax: 100,
252           border: 1,
253           sensitive: ON
254   };
255   accMark1 = {
256           string: "100%",
257           frame:  ([accPedal.box.ne + (1.c, -1.r/2), 1, 4]),
258           alignment: LEFT
259   };
260   accMark2 = {
261           string: "0%",
262           frame:  ([accPedal.box.se + (1.c, -1.r/2), 1, 4]),
263           alignment: LEFT
264   };
265
266   display dispAcc;
267   dispAcc = < accTitle / accPedal / accMark1 / accMark2 >;
268
269   # throttle
270
271   integer Tlength, Twidth;        # for both brake and accelerator
272   point throttleOrg;
273   window throttleTitle, throttleChart, throttleMark1, throttleMark2;
274   Tlength = BAlength/2;
275   Twidth = BAwidth/2;
276   throttleOrg = (225, 530);
277
278   throttleTitle = {
279           string: "Throttle",
280           frame: ([throttleChart.box.n - (10.c / 2, 2.r), 1, 10]),
281           alignment: CENTRE
282   };
283   throttleChart = {
284           type: DONALD,
```

```
285             box: [throttleOrg, throttleOrg + (Twidth, Tlength)],
286             pict: "THROTTLE",
287             xmax: 100,
288             ymax: 100,
289             border: 1
290     };
291     throttleMark1 = {
292             string: "100%",
293             frame:  ([throttleChart.box.nw - (5.c, 1.r/2), 1, 4]),
294             alignment: RIGHT
295     };
296     throttleMark2 = {
297             string: "0%",
298             frame:  ([throttleChart.box.sw - (5.c, 1.r/2), 1, 4]),
299             alignment: RIGHT
300     };
301
302     display dispThrottle;
303     dispThrottle = <throttleTitle / throttleChart / throttleMark1 / throttleMark2>;
304
305
306     # map
307
308     window map;
309     map = {
310             type: DONALD,
311             box: [(290,530), (490,630)],
312             pict: "MAP",
313             border: 1,
314             xmin:-100,
315             xmax:1100,
316             ymin:-10,
317             ymax:100
318     };
319
320     # begin map.d
321     %donald
322     viewport MAP
323     line range1, range2, range3, range4, range5, range6, range7, range8, range9, range10
324     real altitude0, altitude1, altitude2, altitude3, altitude4, altitude5, altitude6, alt
        itude7, altitude8, altitude9, altitude10
325     range1 = [(0,altitude0), (100,altitude1)]
326     range2 = [(100,altitude1), (200,altitude2)]
327     range3 = [(200,altitude2), (300,altitude3)]
328     range4 = [(300,altitude3), (400,altitude4)]
329     range5 = [(400,altitude4), (500,altitude5)]
330     range6 = [(500,altitude5), (600,altitude6)]
331     range7 = [(600,altitude6), (700,altitude7)]
332     range8 = [(700,altitude7), (800,altitude8)]
333     range9 = [(800,altitude8), (900,altitude9)]
334     range10 = [(900,altitude9), (1000,altitude10)]
335     label maptitle
336     maptitle = label("Vehicle Position", (110,70))
337     %eden
338
339     func x2altitude ( para x;
340             return (-cos(x / 1000.0 * 2 * pi) + 1) * 6.3 / 200 * 1000 / 2;
341     )
342     _altitude0 is x2altitude(0);
343     _altitude1 is x2altitude(100);
344     _altitude2 is x2altitude(200);
345     _altitude3 is x2altitude(300);
346     _altitude4 is x2altitude(400);
347     _altitude5 is x2altitude(500);
348     _altitude6 is x2altitude(600);
349     _altitude7 is x2altitude(700);
350     _altitude8 is x2altitude(800);
351     _altitude9 is x2altitude(900);
352     _altitude10 is x2altitude(1000);
353     %scout
354     # end map.d
```



```
355
356     # calculating DoNaLD labels' displacement
357     integer _labelHdisp, _labelVdisp;
358     _labelHdisp = 1.c * (speedometer.xmax - speedometer.xmin) /
359             (speedometer.box.e.1 - speedometer.box.w.1);
360     _labelVdisp =  - 1.r / 2 * (speedometer.ymax - speedometer.ymin) /
361             (speedometer.box.n.2 - speedometer.box.s.2);
362
363     #bridging definitions (EDEN->DoNaLD) for car position
364     %eden
365     _HDistance is int(sampleHDisp) % 1000;
366     _VDistance is sampleVDisp;
367
368     %donald
369
370     viewport MAP
371     label carPos
372     real HDistance, VDistance
373     carPos = label(".", (HDistance, VDistance) + (-20, 3))
374
375     viewport THROTTLE
376     line throttleLine
377     real sampleThrottlePos
378     throttleLine = [(0, sampleThrottlePos * 100), (100, sampleThrottlePos * 100)]
379
380     viewport BRAKE
381     line brakePedal
382     real brakePos
383     %eden
384     _brakePos is brakePos;
385     %donald
386     brakePedal = [(0, brakePos * 100), (100, brakePos * 100)]
387
388     viewport ACC
389     line accelPedal
390     real accelPos
391     %eden
392     _accelPos is accelPos;
393     %donald
394     accelPedal = [(0, accelPos * 100), (100, accelPos * 100)]
395
396     viewport UPARROW
397     line arrow1, arrow2, arrow3
398     arrow1 = [(200,600), (500,800)]
399     arrow2 = [(800,600), (500,800)]
400     arrow3 = [(500,200), (500,800)]
401
402     viewport SPEEDO
403     point ptr
404     line needle
405     real needleLength
406     needleLength = 100.0
407     real minA, maxA, A
408     real pi, ratio
409     real curSpeed, topSpeed
410     real labelHdisp, labelVdisp
411     pi = 3.1416
412     minA = 4 * pi div 3
413     maxA = - pi div 3
414     needle = [(0,0), (needleLength @ A) ]
415     topSpeed = 80.0
416     A = minA + (maxA - minA) * curSpeed div topSpeed
417
418     label L0, L10, L20, L30, L40, L50, L60, L70, L80
419     point P0, P10, P20, P30, P40, P50, P60, P70, P80
420     real A0, A10, A20, A30, A40, A50, A60, A70, A80
421     line mark0, mark10, mark20, mark30, mark40, mark50, mark60, mark70, mark80
422     real gap1, gap2, LSpc
423     gap1, gap2, LSpc = 10.0, 30.0, 50.0
424
425     A0 = minA
```

```
426   P0 = {(needleLength + gap2) @ A0)
427   mark0 = [P0, {(needleLength + gap1) @ A0)]
428   L0 = label("0", P0-(labelHdisp,labelVdisp)+(LSpc @ A0))
429
430   A10 = minA + (maxA - minA) div 8
431   P10 = {(needleLength + gap2) @ A10)
432   mark10 = [P10, {(needleLength + gap1) @ A10)]
433   L10 = label("10", P10 - (labelHdisp, labelVdisp) + (LSpc @ A10))
434
435   A20 = minA + (maxA - minA) * 2 div 8
436   P20 = {(needleLength + gap2) @ A20)
437   mark20 = [P20, {(needleLength + gap1) @ A20)]
438   L20 = label("20", P20 - (labelHdisp, labelVdisp) + (LSpc @ A20))
439
440   A30 = minA + (maxA - minA) * 3 div 8
441   P30 = {(needleLength + gap2) @ A30)
442   mark30 = [P30, {(needleLength + gap1) @ A30)]
443   L30 = label("30", P30 - (labelHdisp, labelVdisp) + (LSpc @ A30))
444
445   A40 = minA + (maxA - minA) * 4 div 8
446   P40 = {(needleLength + gap2) @ A40)
447   mark40 = [P40, {(needleLength + gap1) @ A40)]
448   L40 = label("40", P40 - (labelHdisp, labelVdisp) + (LSpc @ A40))
449
450   A50 = minA + (maxA - minA) * 5 div 8
451   P50 = {(needleLength + gap2) @ A50)
452   mark50 = [P50, {(needleLength + gap1) @ A50)]
453   L50 = label("50", P50 - (labelHdisp, labelVdisp) + (LSpc @ A50))
454
455   A60 = minA + (maxA - minA) * 6 div 8
456   P60 = {(needleLength + gap2) @ A60)
457   mark60 = [P60, {(needleLength + gap1) @ A60)]
458   L60 = label("60", P60 - (labelHdisp, labelVdisp) + (LSpc @ A60))
459
460   A70 = minA + (maxA - minA) * 7 div 8
461   P70 = {(needleLength + gap2) @ A70)
462   mark70 = [P70, {(needleLength + gap1) @ A70)]
463   L70 = label("70", P70 - (labelHdisp, labelVdisp) + (LSpc @ A70))
464
465   A80 = maxA
466   P80 = {(needleLength + gap2) @ A80)
467   mark80 = [P80, {(needleLength + gap1) @ A80)]
468   L80 = label("80", P80 - (labelHdisp, labelVdisp) + (LSpc @ A80))
469
470   %eden
471
472   proc Send3 { para message;
473       auto ok;
474       ok = send_msg(stdmsg, [3, message], 0);
475       if (ok == -1) error("can't write to message queue");
476   }
477
478   nextClock = iClock;
479   proc clocking : iClock, stopCLK {
480           if (!stopCLK)
481           {
482                   nextClock++;
483                   Send3("iClock = nextClock;\n");
484           }
485           /*
486                   Send3("iClock = "//str(iClock+1)//";\n");
487           */
488   }
489
490   proc stopClk: clkStopBtn_mouse_1 {
491           if (clkStopBtn_mouse_1[2]==4) stopCLK = 1;
492   }
493
494   proc startClk : clkStartBtn_mouse_1 {
495           if (clkStartBtn_mouse_1[2]==4) {
496                   stopCLK = 0;
```



```
497           }
498   }
499
500   proc resetClk : clkResetBtn_mouse_1 {
501           if (clkResetBtn_mouse_1[2]==4) {
502                   iClock = nextClock = 0;
503           }
504   }
505
506   proc chgEngineStts : ignition_mouse_1 {
507           if (ignition_mouse_1[2]==4) {
508                   engineStts = (engineStts == esOn) ? esOff : esOn;
509           }
510   }
511
512   proc chgBrakePos : brakePedal_mouse {
513           if (brakePedal_mouse[2]==4) {
514                   brakePos = brakePedal_mouse[5] / 100.0;
515           }
516   }
517
518   proc chgAccelPos : accPedal_mouse {
519           if (accPedal_mouse[2]==4) {
520                   accelPos = accPedal_mouse[5] / 100.0;
521           }
522   }
523
524   incrBtn = pbUp;
525   proc incCrSpeed : incrBtn, crUpWin_mouse {
526           if (crUpWin_mouse[2] == 4) {
527                   Send3("incrBtn = pbDown;\n");
528           } else {
529                   if (incrBtn == pbDown)
530                           Send3("incrBtn = pbUp;\n");
531           }
532           if ((cruiseStts != csOff) && (cruiseSpeed_mph < maxCruiseSpeed_mph)) {
533                   cruiseSpeed_mph = cruiseSpeed_mph + 1;
534           }
535   }
536
537   decrBtn = pbUp;
538   proc decCrSpeed : decrBtn, crDownWin_mouse {
539           if (crDownWin_mouse[2] == 4) {
540                   Send3("decrBtn = pbDown;\n");
541           } else {
542                   if (decrBtn == pbDown)
543                           Send3("decrBtn = pbUp;\n");
544           }
545           if ((cruiseStts != csOff) && (cruiseSpeed_mph > minCruiseSpeed_mph)) {
546                   cruiseSpeed_mph = cruiseSpeed_mph - 1;
547           }
548   }
549
550   proc cruiseOn : crOnBtn_mouse_1 {
551           if (crOnBtn_mouse_1[2] == 4 && onBtn_prev == pbUp) {
552                   onBtn = pbDown;
553           }
554   }
555
556   proc cruiseOff : crOffBtn_mouse_1 {
557           if (crOffBtn_mouse_1[2] == 4 && offBtn_prev == pbUp) {
558                   offBtn = pbDown;
559           }
560   }
561
562   setBtn is (crSetBtn_mouse_1[2] == 4) ? pbDown : pbUp;
563   resBtn is (crResumeBtn_mouse_1[2] == 4) ? pbDown : pbUp;
564   manBtn is (crManualBtn_mouse_1[2] == 4) ? pbDown : pbUp;
565
566   %scout
567   window vehicle;
```

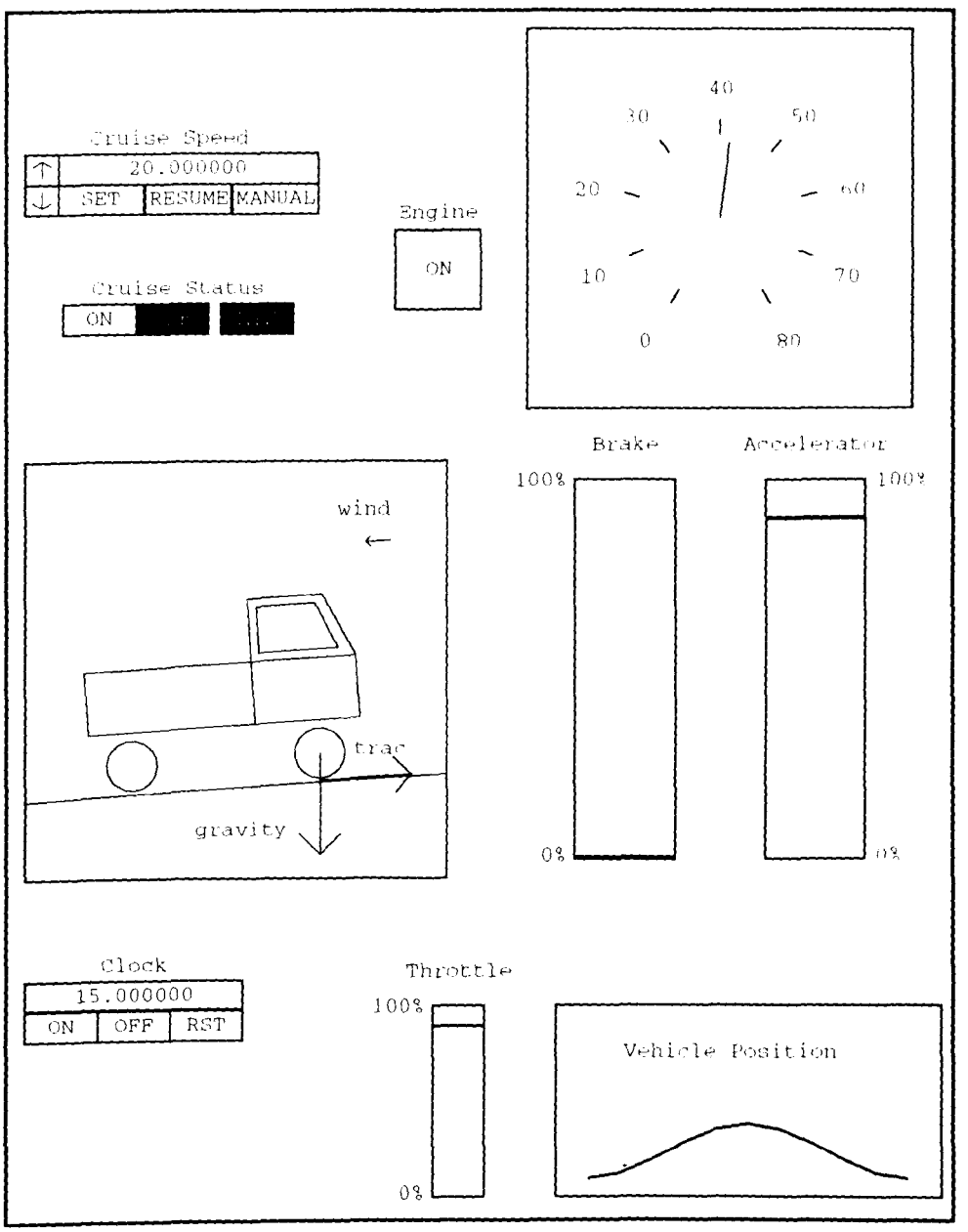
```
568   vehicle = (
569           type:   DONALD,
570           box:    [(10, 240), (230, 460)],
571           pict:   "VEHICLE",
572           xmin:   -700,
573           xmax:   300,
574           ymin:   -300,
575           ymax:   700,
576           border: 1
577   );
578
579   %donald
580   viewport IMAGINARY
581   openshape conceptCar
582   within conceptCar (
583           circle frontWheel, backWheel
584           frontWheel = circle((0,0), 60)
585           backWheel = circle((-450, 0), 60)
586   #       backWheel = circle((200-fullLength, 0), 60)
587           point vp1, vp2, vp3, vp4, vp5, vp6, vp7, vp8, vp9, vp10, vp11
588           line vl1, vl2, vl3, vl4, vl5, vl6, vl7, vl8, vl9, vl10, vl11
589           int height, frontLength, fullLength, roofLength, gap
590           height = 300
591           frontLength = 250
592           fullLength = 650
593           roofLength = 180
594           gap = 20
595           vp1 = (100, 80)
596           vp2 = vp1 + (0, height div 2)
597           vp3 = vp2 - (fullLength, 0)
598           vp4 = vp1 - (fullLength, 0)
599           vp5 = vp1 - (frontLength, 0)
600           vp6 = vp5 + (0, height)
601           vp7 = vp6 + (roofLength, 0)
602           vp8 = vp2 + (-2 * gap, gap)
603           vp9 = vp7 - (gap, gap)
604           vp10 = vp6 + (gap, -gap)
605           vp11 = vp2 - (frontLength, 0) + (gap, gap)
606           vl1 = [vp1, vp2]
607           vl2 = [vp2, vp3]
608           vl3 = [vp3, vp4]
609           vl4 = [vp1, vp4]
610           vl5 = [vp5, vp6]
611           vl6 = [vp2, vp7]
612           vl7 = [vp6, vp7]
613           vl8 = [vp8, vp9]
614           vl9 = [vp9, vp10]
615           vl10 = [vp10, vp11]
616           vl11 = [vp11, vp8]
617   )
618
619   openshape arrow
620   within arrow (
621           line arrowBody, arrowHead1, arrowHead2
622           arrowBody = [(0,0), (1,0)]
623           arrowHead1 = [(1,0), (0.8,0.2)]
624           arrowHead2 = [(1,0), (0.8,-0.2)]
625   )
626
627   viewport VEHICLE
628   real gradient, windF, gravF, tracF, brakF
629
630   shape vehicle
631   vehicle = rot(conceptCar, (0,0), gradient)
632
633   line ground
634   ground = rot([(-1000, -70), (500, -70)], (0,0), gradient)
635
636   shape wind
637   wind = rot(trans(scale(arrow, -windF div 30), 200, 500), (0,0), gradient)
638   label windLabel
```



```
639   windLabel = label("wind", rot((80, 570), (0,0), gradient))
640
641   shape gravity
642   gravity = rot(scale(arrow, gravF div 100), (0,0), -pi div 2)
643   label gravLabel
644   gravLabel = label("gravity", (-300, -200))
645
646   shape tracting
647   tracting = rot(trans(scale(arrow, tracF div 30), 0, -65), (0,0), gradient)
648   label tracLabel
649   tracLabel = label(if (tracF != 0) then "trac" else "", \
650                   rot((80, -10), (0,0), gradient))
651
652   shape brakeForce
653   brakeForce = rot(trans(scale(arrow, -brakF div 30), 0, -65), (0,0), gradient)
654   label brakeLabel
655   brakeLabel = label(if (brakF != 0) then "brake" else "", \
656                   rot((-270, -10), (0,0), gradient))
657
658   %scout
659   screen = dispClk & dispCrCnt & <speedometer> & <map> &
660            dispBrake & dispAcc & dispThrottle & dispEngine & dispCrSpd &
661            <vehicle>;
```

The Vehicle Cruise Control Simulation Example

G.4. Sample Output


Cruise Speed
20.000000
SET
RESUME
MANUAL
Engine
ON
Cruise Status
ON
40
30
50
20
60
10
70
0
80
wind
trac
gravity
Brake
Accelerator
100%
100%
0%
0%
Clock
15.000000
ON
OFF
RST
Throttle
100%
0%
Vehicle Position

Appendix H

The Railway Station Simulation Example

H.1. The LSD Specification

H.2. A Corresponding ADM Program

H.3. EDEN Implementation of the ADM Program

H.4. Extract of a Textual Simulation

H.5. Scout Graphical Interface of the Simulation

H.6. A Sample of the Graphical Output

The Railway Station Simulation Example

H.1. The LSD Specification

```
agent sm() {                                    // The station master:
state     (time) tarrive = |Time|;              // registers time of arrival
          (bool) can_move = false;              // determines whether the driver can start the engine
          (bool) whistle = false;               // controls the whistle
          (bool) whistled = false;              // remembers whether he has blown the whistle
          (bool) sm_flag = false;               // controls the flag
          (bool) sm_raised_flag = false;        // remembers whether he has raised the flag
oracle    (time) Limit, Time;                   // knows the time to elapse before departure due
          (bool) guard_raised_flag;             // knows whether the guard has raised his flag
          (bool) driver_ready;                  // knows the driver is ready
          (bool) around[d]; (d = 1 .. number_of_doors)
                                                // knows whether there's anybody around doorway
          (bool) door_open[d]; (d = 1 .. number_of_doors)  // the doors status
handle    (bool) can_move, whistle, whistled, sm_flag, sm_raised_flag;
          (bool) door_open[d]; (d = 1 .. number_of_doors)  // partially controls the doors

derive    (bool) ready = ∧ (¬door_open[d]) | d = 1 .. number_of_doors);
                                                // monitors whether all doors are shut
          (bool) timeout = (Time – tarrive) > Limit;       // monitors whether departure is due
privilege door_open[d] ^ ¬around[d] –> door_open[d] = false; (d = 1 .. number_of_doors)
          ready ^ timeout ^ ¬whistled –> whistle = true; whistled = true; guard(); whistle = false;
          ready ^ whistled ^ ¬sm_raised_flag –> sm_flag = true; sm_raised_flag = true;
          sm_flag ^ guard_raised_flag –> sm_flag = false;
          ready ^ guard_raised_flag ^ driver_ready ^ engaged ^ ¬can_move –> can_move = true;
}

agent guard() {
state     (bool) guard_raised_flag = false; guard_flag = false;
oracle    (bool) engaging, whistled, guard_flag, sm_flag, sm_raised_flag, brake;
handle    (bool) brake, guard_flag, guard_raised_flag;
derive    LIVE = engaging || whistled;
privilege engaging ^ ¬brake –> brake = true;
          sm_raised_flag ^ brake –> brake = false; guard_flag = true; guard_raised_flag = true;
          guard_flag ^ ¬sm_flag –> guard_flag = false;
}

agent driver() {
state     (bool) driver_ready = false;
oracle    (bool) can_move, engaged, whistled;
          (int) at, from;
handle    (int) from, to;
          (bool) driver_ready, running;
privilege engaged ^ whistled ^ ¬driver_ready –> driver_ready = true;
          engaged ^ from <> at –> from = | at |; to = 3 - from;
          engaged ^ can_move –> driver_ready = false; running = true;
}

agent train() {
state     (bool) running = true; brake = false; alarm = false
          (int) from = 0; to = 1; at = 1;
oracle    (bool) alarm, brake, running;
          (int) from, to, at;
handle    (bool) running, alarm;
derive    (bool) engaging = running ^ to == at;
          (bool) leaving = running ^ from == at;
          (bool) engaged = ¬running;
privilege engaging ^ ¬alarm –> alarm = true; guard(); sm();
          leaving ^ alarm –> alarm = false; delete guard(), sm();
          brake ^ running –> running = false;
}
```

Figure 1

```
agent passenger((int) p, (int) d, (int) _from, (int) _to) {
// passenger p is intending to travel from station _from to station _to
// and he will access through door d of the train
state       (int) from[p] = _from;
            (int) to[p] = _to;
            (int) pat[p] = _from;
            (int) door[p] = d;
            (int) pos[p] = 2;
            (bool) alighting[p], boarding[p], join_queue[p,d];
oracle      (int) at, pat[p];
            (bool) queueing[d], pos[p], door_open[d];
handle      (int) pos[p], pat[p];
            (bool) door_open[d];
derivate    alighting[p] = at == pat[p] ^ at == to[p] ^ -2 ≤ pos[p] ≤ 0 ^ engaged;
            boarding[p] = at == pat[p] ^ at == from[p] ^ 0 ≤ pos[p] ≤ 2 ^ engaged;
            join_queue[p,d] = (alighting[p] ^ door_open[d] ^ pos[p] == -1) ||
                              (boarding[p] ^ door_open[d] ^ pos[p] == 1);
            LIVE = ¬(pat[p] == to[p] ^ pos[p] == 2);
privilege   boarding[p] ^ pos[p] == 2 -> pos[p] = 1;
            alighting[p] ^ pos[p] == -2 -> pos[p] = -1;
            alighting[p] ^ ¬door_open[d] -> door_open[d] = true;
            alighting[p] ^ pos[p] == 0 ^ door_open[d] ^ queuing[d]
                              -> pos[p] = 1; pat[p] = lat|; pos[p] = 2;
            alighting[p] ^ pos[p] == 0 ^ door_open[d] ^ ¬queuing[d]
                              -> pos[p] = 1; pat[p] = lat|; door_open[d] = false; pos[p] = 2;
            boarding[p] ^ ¬door_open[d] -> door_open[d] = true;
            boarding[p] ^ pos[p] == 0 ^ door_open[d] ^ queuing[d]
                              -> pos[p] = -1; pat[p] = at; pos[p] = -2;
            boarding[p] ^ pos[p] == 0 ^ door_open[d] ^ ¬queuing[d]
                              -> pos[p] = -1; pat[p] = at; door_open[d] = false; pos[p] = -2;
}

agent door((int) d) {
state       (bool) queuing[d], occupied[d], around[d];
            (bool) door_open[d] = false;
oracle      (int) pos[p], door[p]; (p = 1 .. number_of_passengers)
            (bool) join_queue[p,d]; (p = 1 .. number_of_passengers)
handle      (int) pos[p]; (p = 1 .. number_of_passengers)
derivate    queuing[d] = there exists p such that join_queue[p,d] == true;
            occupied[d] = there exists p such that (pos[p] == 0 ^ door[p] == d)
            around[d] = there exists p such that (door[p] == d ^ -1 ≤ pos[p] ≤ 1)
privilege   queuing[d] ^ ¬occupied[d] ^ join_queue[p,d] -> pos[p] = 0; (p = 1 .. number_of_passengers)
}
```

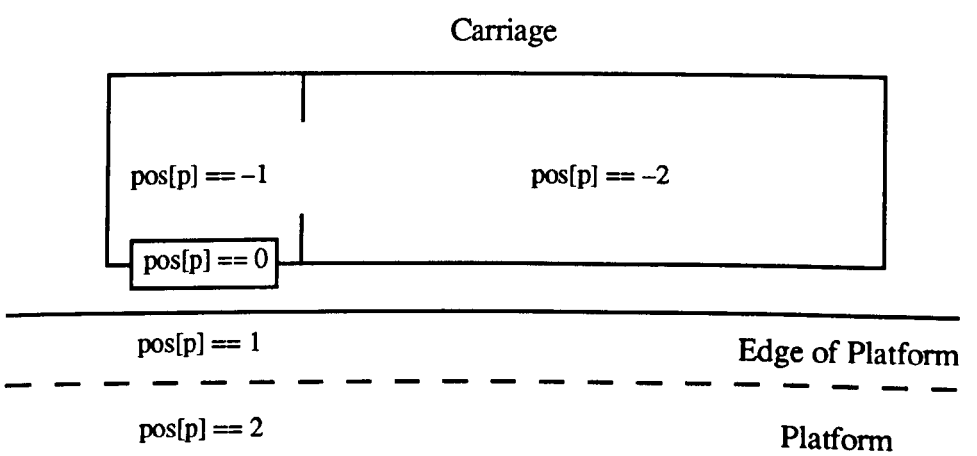


Figure 2

The Railway Station Simulation Example

H.2. A Corresponding ADM Program

```
entity passenger(_p, _d, _from, _to) {
definition
        from[_p] = _from,
        to[_p] = _to,
        pat[_p] = _from,
        door[_p] = _d,
        pos[_p] = 1,
        alighting[_p] = at == pat[_p] && at == to[_p] && pos[_p] <= 0 && engaged,
        boarding[_p] = at == pat[_p] && at == from[_p] && pos[_p] >= 0 && engaged,
        join_queue[_p,_d] = (alighting[_p] && door_open[_d] && pos[_p] == -1) ||
                            (boarding[_p] && door_open[_d] && pos[_p] == 1),
        join_queue[_p,3-_d] = false,
        state[_p] = 0
action
        pat[_p] == to[_p] && pos[_p] == 2
            -> delete passenger(_p, _d, _from, _to),
        boarding[_p] && pos[_p] == 2
            print("Passenger ",_p," goes to the edge of platform")
                -> pos[_p] = 1,
        alighting[_p] && pos[_p] == -2
            print("Passenger ",_p," goes near door ",_d)
                -> pos[_p] = -1,
        alighting[_p] && !door_open[_d] && rand(6) == 1
            print("Passenger ",_p," opens door ",_d)
                -> door_open[_d] = true,
        alighting[_p] && pos[_p] == 0 && door_open[_d] && !queuing[_d]
            print("Passenger ",_p," alighting on platform")
                -> pos[_p] = 1; state[_p] = 1; pat[_p] = |at|,
        state[_p] == 1 && door_open[_d] && !queuing[_d]
            print("Passenger ",_p," closes door ",_d)
                -> door_open[_d] = false; state[_p] = 2,
        state[_p] == 2
            print("Passenger ",_p," leaves the station")
                -> pos[_p] = 2; state[_p] = 0,
        alighting[_p] && pos[_p] == 0 && door_open[_d] && queuing[_d]
            print("Passenger ",_p," alighting on platform")
                -> pos[_p] = 1; state[_p] = 2,
        boarding[_p] && !door_open[_d] && rand(6) == 1
            print("Passenger ",_p," opens door ",_d)
                -> door_open[_d] = true,
        boarding[_p] && pos[_p] == 0 && door_open[_d] && !queuing[_d]
            print("Passenger ",_p," entering the train")
                -> pos[_p] = -1; state[_p] = 4; pat[_p] = at,
        state[_p] == 4 && door_open[_d] && !queuing[_d]
            print("Passenger ",_p," closes door ",_d)
                -> door_open[_d] = false; state[_p] = 5,
        state[_p] == 5
            print("Passenger ",_p," takes a seat")
                -> pos[_p] = -2; state[_p] = 0,
        boarding[_p] && pos[_p] == 0 && door_open[_d] && queuing[_d]
            print("Passenger ",_p," entering the train")
                -> pos[_p] = -1; state[_p] = 5
}

entity door(_d) {
definition
        queuing[_d] = join_queue[1,_d] || join_queue[2,_d] || join_queue[3,_d],
        occupied[_d] = (pos[1] == 0 && door[1] == _d)
                    || (pos[2] == 0 && door[2] == _d)
                    || (pos[3] == 0 && door[3] == _d),
        around[_d] = ((pos[1] >= -1 && pos[1] <= 1) && door[1] == _d)
                  || ((pos[2] >= -1 && pos[2] <= 1) && door[2] == _d)
                  || ((pos[3] >= -1 && pos[3] <= 1) && door[3] == _d),
        only_one[_d] = |rand(3)|
action
        only_one[_d] == 1 && queuing[_d] && !occupied[_d] && join_queue[1,_d]
            print("Passenger 1 is at the doorway")
                -> pos[1] = 0,
        only_one[_d] == 2 && queuing[_d] && !occupied[_d] && join_queue[2,_d]
            print("Passenger 2 is at the doorway")
                -> pos[2] = 0,
```



```
        only_one[_d] == 3 && queuing[_d] && !occupied[_d] && join_queue[3,_d]
            print("Passenger 3 is at the doorway")
                -> pos[3] = 0,
        true
                -> only_one[_d] = |rand(3)|
}

entity sm() {
definition
        whistle = false,
        whistled = false,
        sm_flag = false,
        sm_raised_flag = false,
        can_move = false,
        ready =  !door_open[1] && !door_open[2],
        tarrive,
        timeout = (time - tarrive) > Limit,
        level = 0,
        init = true
action
        init
                -> tarrive = |time|; init = false,
        door_open[1] && !around[1]
            print("Station master shuts door 1")
                -> door_open[1] = false,
        door_open[2] && !around[2]
            print("Station master shuts door 2")
                -> door_open[2] = false,
        ready && timeout && !whistled
            print("Station master whistles to call guard")
                -> whistle = true; whistled = true; guard(); level = 1,
        level == 1
            print("Station master stops whistling")
                -> whistle = false; level = 0,
        ready && whistled && !sm_raised_flag
            print("Station master raises his flag")
                -> sm_flag = true; sm_raised_flag = true,
        sm_flag && guard_raised_flag
            print("Station master lowers his flag")
                -> sm_flag = false,
        ready && guard_raised_flag && driver_ready && engaged && !can_move
            print("Train can move now")
                -> can_move = true
}

entity guard() {
definition
        guard_raised_flag = false,
        guard_flag = false,
        step = 0
action
        !(engaging || whistled)
            print("Guard is having a tea break")
                -> delete guard(),
        engaging && !brake
            print("Guard applies brake")
                -> brake = true,
        sm_raised_flag && brake
            print("Guard releases brake")
                -> brake = false; step = 1,
        step == 1
            print("Guard raises his flag")
                -> guard_flag = true; guard_raised_flag = true; step = 0,
        guard_flag && !sm_flag
            print("Guard lowers his flag")
                -> guard_flag = false
}

entity driver() {
definition
        driver_ready = false
```

```
143  action
144          engaged && whistled && !driver_ready
145              print("Driver is ready to start engine")
146                  -> driver_ready = true,
147          engaged && from != at
148              print("Changing destination of the train")
149                  -> from = |at|; to = 3 - from,
150          engaged && can_move
151              print("Engine starts ")
152                  -> driver_ready = false; running = true
153  }
154
155  entity train() {
156  definition
157          running = true,
158          brake = false,
159          door_open[1] = false,
160          door_open[2] = false,
161          from = 0,
162          to = 1,
163          at = 1,
164          engaging = running && to == at,
165          leaving = running && from == at,
166          engaged = !running,
167          alarm = false
168  action
169          engaging && !alarm
170              print("Guard and station master finish their tea breaks")
171                  -> guard(); sm(); alarm = true,
172          leaving && alarm
173              print("Guard and station master off duty")
174                  -> delete guard(); delete sm(); alarm = false; at = |3-at|,
175          brake && running
176              print("The train stops")
177                  -> running = false
178  }
179
180  entity initialiser() {
181  definition
182          Limit = 20,
183          time = 0
184  action
185          true
186                  -> time = |time| + 1
187  }
188
189  door(1) door(2) passenger(1,1,1,2) passenger(2,1,1,2) passenger(3,1,1,2)
190  driver() initialiser() train()
191
```

The Railway Station Simulation Example

H.3. EDEN Implementation of the ADM Program

```
1     func isTrue ( para v;
2             return v != @ && v;
3     )
4
5     func isDefined ( para v;
6             return v != @;
7     )
8
9     func Rand ( para i;
10            return rand() % i + 1;
11    )
12
13    proc stop {
14            stopClock = 1;
15            if (at == 1)
16                    SendToEden("at = 2; nextClock += 50 - 1; startClock = 1;\n");
17            else
18                    SendToEden("at = 1; nextClock += 50 - 1; startClock = 1;\n");
19    }
20
21    true = 1;
22    false = 0;
23
24    /*-------------------------------------------------------------------------
25    function macro(macro_str, para_str1, para_str2, .., para_strN)
26      Expands 'macro_str' by substituting 'para_strI' for "?I" and returns the
27      resultant string (ref.: Edward Yung, M.Sc. thesis, '89, vol.2).
28    -------------------------------------------------------------------------*/
29
30    func macro {
31      auto i, j, l, m, n, c, s;
32      s = "";
33      l = (m = $1)#;
34      shift;
35      i = 1;
36      while (i <= l) {
37        for (j = i; j <= l && m[j] != '?'; j++);
38        if (i != j) s = s // substr(m, i, j - 1);
39        if (j <= l) {
40          j++;
41          n = (c = (j > l) ? '?' : m[j]) - '0';
42          s = s // ((1 <= n && n <= $#) ? $[n] : c);
43        }
44        i = j + 1;
45      }
46      return s;
47    }
48
49    proc init_passenger ( para p, d, from, to;
50    writeln("instantiating passenger ", p, " ", d, " ", from, " ", to);
51    execute(macro("
52    from_?1 = ?3;
53    to_?1 = ?4;
54    pat_?1 = ?3;
55    door_?1 = ?2;
56    pos_?1 = 2;
57    alighting_?1 is at == pat_?1 && at == to_?1 && pos_?1 <= 0 && isTrue(engaged);
58    boarding_?1 is at == pat_?1 && at == from_?1 && pos_?1 >= 0 && isTrue(engaged);
59    join_queue_?1_?2 is (alighting_?1 && door_open_?2 && pos_?1 == -1) ||
60                        (boarding_?1 && door_open_?2 && pos_?1 == 1);
61    join_queue_?1_"//str(3-d)//" is false;
62    state_?1 is 0;
63    "//"
64    proc passenger_?1_live : sysClock {
65    if (sysClock == -1) return;
66            if (pat_?1 == to_?1 && pos_?1 == 2) {
67                    SendToEden(\"delete_passenger(?1, ?2, ?3, ?4);\n\");
68            }
69    }
70    "//"
71    proc passenger_?1_action_1 : sysClock {
```



```
72    if (sysClock == -1) return;
73            if (boarding_?1 && pos_?1 == 2) {
74                    writeln(\"Passenger ?1 goes to the edge of the platform\");
75                    SendToEden(\"pos_?1 = 1;\n\");
76            }
77    }
78    "//"
79    proc passenger_?1_action_2 : sysClock {
80    if (sysClock == -1) return;
81            if (alighting_?1 && pos_?1 == -2) {
82                    writeln(\"Passenger ?1 goes near door ?2\");
83                    SendToEden(\"pos_?1 = -1;\n\");
84            }
85    }
86    "//"
87    proc passenger_?1_action_3 : sysClock {
88    if (sysClock == -1) return;
89            if (alighting_?1 && !door_open_?2 && Rand(6) == 1) {
90                    writeln(\"Passenger ?1 opens door ?2\");
91                    SendToEden(\"door_open_?2 = true;\n\");
92            }
93    }
94    "//"
95    proc passenger_?1_action_4 : sysClock {
96    if (sysClock == -1) return;
97            if (alighting_?1 && pos_?1 == 0 && door_open_?2 && !isTrue(queuing_?2)) {
98                    writeln(\"Passenger ?1 alighting on platform\");
99                    SendToEden(\"pat_?1 = at; pos_?1 = 1; state_?1 = 1;\n\");
100           }
101   }
102   "//"
103   proc passenger_?1_action_5 : sysClock {
104   if (sysClock == -1) return;
105           if (state_?1 == 1 && door_open_?2 && !isTrue(queuing_?2)) {
106                   writeln(\"Passenger ?1 closes door ?2\");
107                   SendToEden(\"door_open_?2 = false; state_?1 = 2;\n\");
108           }
109   }
110   "//"
111   proc passenger_?1_action_6 : sysClock {
112   if (sysClock == -1) return;
113           if (state_?1 == 2) {
114                   writeln(\"Passenger ?1 leaves the station\");
115                   SendToEden(\"pos_?1 = 2; state_?1 = 0;\n\");
116           }
117   }
118   "//"
119   proc passenger_?1_action_7 : sysClock {
120   if (sysClock == -1) return;
121           if (alighting_?1 && pos_?1 == 0 && door_open_?2 && isTrue(queuing_?2)) {
122                   writeln(\"Passenger ?1 alighting on platform\");
123                   SendToEden(\"pat_?1 = at; pos_?1 = 1; state_?1 = 2;\n\");
124           }
125   }
126   "//"
127   proc passenger_?1_action_8 : sysClock {
128   if (sysClock == -1) return;
129           if (boarding_?1 && !door_open_?2 && Rand(6) == 1) {
130                   writeln(\"Passenger ?1 opens door ?2\");
131                   SendToEden(\"door_open_?2 = true;\n\");
132           }
133   }
134   "//"
135   proc passenger_?1_action_9 : sysClock {
136   if (sysClock == -1) return;
137           if (boarding_?1 && pos_?1 == 0 && door_open_?2 && !isTrue(queuing_?2)) {
138                   writeln(\"Passenger ?1 entering the train\");
139                   SendToEden(\"pat_?1 is at; pos_?1 = -1; state_?1 = 4;\n\");
140           }
141   }
142   "//"
```

```
143  proc passenger_?1_action_10 : sysClock {
144  if (sysClock == -1) return;
145          if (state_?1 == 4 && door_open_?2 && !isTrue(queuing_?2)) {
146                  writeln(\"Passenger ?1 closes door ?2\");
147                  SendToEden(\"door_open_?2 = false; state_?1 = 5;\n\");
148          }
149  }
150  "//"
151  proc passenger_?1_action_11 : sysClock {
152  if (sysClock == -1) return;
153          if (state_?1 == 5) {
154                  writeln(\"Passenger ?1 takes a seat\");
155                  SendToEden(\"pos_?1 = -2; state_?1 = 0;\n\");
156          }
157  }
158  "//"
159  proc passenger_?1_action_12 : sysClock {
160  if (sysClock == -1) return;
161          if (boarding_?1 && pos_?1 == 0 && door_open_?2 && isTrue(queuing_?2)) {
162                  writeln(\"Passenger ?1 entering the train\");
163                  SendToEden(\"pat_?1 is at; pos_?1 = -1; state_?1 = 5;\n\");
164          }
165  }
166  ", str(p), str(d), str(from), str(to)));
167  }
168
169  proc gen_passenger { para p;
170          auto from;
171          from = Rand(2);
172          init_passenger(p, Rand(2), from, 3-from);
173  }
174
175  proc delete_passenger { para p, d, from, to;
176  writeln("delete passenger ", p);
177  execute(macro("
178  forget(\"passenger_?1_live\");
179  forget(\"passenger_?1_action_1\");
180  forget(\"passenger_?1_action_2\");
181  forget(\"passenger_?1_action_3\");
182  forget(\"passenger_?1_action_4\");
183  forget(\"passenger_?1_action_5\");
184  forget(\"passenger_?1_action_6\");
185  forget(\"passenger_?1_action_7\");
186  forget(\"passenger_?1_action_8\");
187  forget(\"passenger_?1_action_9\");
188  forget(\"passenger_?1_action_10\");
189  forget(\"passenger_?1_action_11\");
190  forget(\"passenger_?1_action_12\");
191  join_queue_?1_1 = @;
192  join_queue_?1_2 = @;
193  alighting_?1 = @;
194  boarding_?1 = @;
195  pat_?1 = @;
196  from_?1 = @;
197  to_?1 = @;
198  door_?1 = @;
199  pos_?1 = @;
200  gen_passenger(?1);
201  ", str(p), str(d), str(from), str(to)));
202  }
203
204  proc init_door { para d;
205  writeln("instantiating door ", d);
206  execute(macro("
207  queuing_?1 is isTrue(join_queue_1_?1)
208              || isTrue(join_queue_2_?1)
209              || isTrue(join_queue_3_?1);
210  occupied_?1 is (isDefined(pos_1) && isDefined(door_1) && pos_1==0 &&door_1==?1)
211              || (isDefined(pos_2) && isDefined(door_2) && pos_2==0 &&door_2==?1)
212              || (isDefined(pos_3) && isDefined(door_3) && pos_3==0 &&door_3==?1);
213  around_?1 is (isDefined(pos_1) && isDefined(door_1) &&
```



```
214                  (pos_1 >= -1 && pos_1 <= 1) && door_1 == ?1)
215              || (isDefined(pos_2) && isDefined(door_2) &&
216                  (pos_2 >= -1 && pos_2 <= 1) && door_2 == ?1)
217              || (isDefined(pos_3) && isDefined(door_3) &&
218                  (pos_3 >= -1 && pos_3 <= 1) && door_3 == ?1);
219  only_one_?1 = Rand(3);
220  "//"
221  proc door_?1_action_1 : sysClock {
222  if (sysClock == -1) return;
223          if (only_one_?1 == 1 && queuing_?1 && !occupied_?1 && isTrue(join_queue_1_?1)
     ) {
224                  writeln(\"Passenger 1 is at the doorway\");
225                  SendToEden(\"pos_1 = 0;\n\");
226          }
227  }
228  "//"
229  proc door_?1_action_2 : sysClock {
230  if (sysClock == -1) return;
231          if (only_one_?1 == 2 && queuing_?1 && !occupied_?1 && isTrue(join_queue_2_?1)
     ) {
232                  writeln(\"Passenger 2 is at the doorway\");
233                  SendToEden(\"pos_2 = 0;\n\");
234          }
235  }
236  "//"
237  proc door_?1_action_3 : sysClock {
238  if (sysClock == -1) return;
239          if (only_one_?1 == 3 && queuing_?1 && !occupied_?1 && isTrue(join_queue_3_?1)
     ) {
240                  writeln(\"Passenger 3 is at the doorway\");
241                  SendToEden(\"pos_3 = 0;\n\");
242          }
243  }
244  "//"
245  proc door_?1_action_4 : sysClock {
246  if (sysClock == -1) return;
247          only_one_?1 = Rand(3);
248  }
249  ", str(d)));
250  }
251
252  proc delete_door { para d;
253  writeln("delete door ", d);
254  execute(macro("
255  forget(\"door_?1_action_1\");
256  forget(\"door_?1_action_2\");
257  forget(\"door_?1_action_3\");
258  forget(\"door_?1_action_4\");
259  queuing_?1 = @;
260  occupied_?1 = @;
261  around_?1 = @;
262  only_one_?1 = @;
263  ", str(d)));
264  }
265
266  proc init_sm {
267  writeln("instantiating sm");
268  execute("
269  whistle is false;
270  whistled is false;
271  sm_flag is false;
272  sm_raised_flag is false;
273  can_move is false;
274  ready is !isTrue(door_open_1) && !isTrue(door_open_2);
275  tarrive = @;
276  timeout is isDefined(tarrive) && (sysClock - tarrive) > Limit;
277  level is 0;
278  init is true;
279  "//"
280  proc sm_action_1 : sysClock {
281  if (sysClock == -1) return;
```

```
282             if (init) {
283                     SendToEden(\"tarrive = \"//str(sysClock)//\"; init = false;\n\");
284             }
285     }
286     "//"
287     proc sm_action_2 : sysClock {
288     if (sysClock == -1) return;
289             if (isTrue(door_open_1) && !isTrue(around_1)) {
290                     writeln(\"Station master shuts door 1\");
291                     SendToEden(\"door_open_1 = false;\n\");
292             }
293     }
294     "//"
295     proc sm_action_3 : sysClock {
296     if (sysClock == -1) return;
297             if (isTrue(door_open_2) && !isTrue(around_2)) {
298                     writeln(\"Station master shuts door 2\");
299                     SendToEden(\"door_open_2 = false;\n\");
300             }
301     }
302     "//"
303     proc sm_action_4 : sysClock {
304     if (sysClock == -1) return;
305             if (ready && timeout && !whistled) {
306                     writeln(\"Station master whistles to call guard\");
307                     SendToEden(\"whistle = true; whistled = true; init_guard(); level = 1
    ;\n\");
308             }
309     }
310     "//"
311     proc sm_action_5 : sysClock {
312     if (sysClock == -1) return;
313             if (level == 1) {
314                     writeln(\"Station master stops whistling\");
315                     SendToEden(\"whistle = false; level = 0;\n\");
316             }
317     }
318     "//"
319     proc sm_action_6 : sysClock {
320     if (sysClock == -1) return;
321             if (ready && whistled && !sm_raised_flag) {
322                     writeln(\"Station master raises his flag\");
323                     SendToEden(\"sm_flag = true; sm_raised_flag = true;\n\");
324             }
325     }
326     "//"
327     proc sm_action_7 : sysClock {
328     if (sysClock == -1) return;
329             if (sm_flag && isTrue(guard_raised_flag)) {
330                     writeln(\"Station master lowers his flag\");
331                     SendToEden(\"sm_flag = false;\n\");
332             }
333     }
334     "//"
335     proc sm_action_8 : sysClock {
336     if (sysClock == -1) return;
337             if (ready && isTrue(guard_raised_flag) && isTrue(driver_ready)
338                     && isTrue(engaged) && !can_move) {
339                     writeln(\"Train can move now\");
340                     SendToEden(\"can_move = true;\n\");
341             }
342     }
343     ");
344     }
345
346     proc delete_sm {
347     writeln("delete sm");
348     execute("
349     forget(\"sm_action_1\");
350     forget(\"sm_action_2\");
351     forget(\"sm_action_3\");
```



```
352     forget(\"sm_action_4\");
353     forget(\"sm_action_5\");
354     forget(\"sm_action_6\");
355     forget(\"sm_action_7\");
356     forget(\"sm_action_8\");
357     ready = @;
358     timeout = @;
359     whistle = @;
360     whistled = @;
361     sm_flag = @;
362     sm_raised_flag = @;
363     can_move = @;
364     tarrive = @;
365     level = @;
366     init = @;
367     ");
368     }
369
370     proc init_guard {
371     writeln("instantiating guard");
372     execute("
373     guard_raised_flag = false;
374     guard_flag = false;
375     step = 0;
376     proc guard_action_1 : sysClock {
377     if (sysClock == -1) return;
378             if (!(isTrue(engaging) || isTrue(whistled))) {
379                     writeln(\"Guard is having a tea break\");
380                     SendToEden(\"delete_guard();\n\");
381             }
382     }
383     proc guard_action_2 : sysClock {
384     if (sysClock == -1) return;
385             if (isTrue(engaging) && !isTrue(brake)) {
386                     writeln(\"Guard applies brake\");
387                     SendToEden(\"brake = true;\n\");
388             }
389     }
390     proc guard_action_3 : sysClock {
391     if (sysClock == -1) return;
392             if (isTrue(sm_raised_flag) && isTrue(brake)) {
393                     writeln(\"Guard releases brake\");
394                     SendToEden(\"brake = false; step = 1;\n\");
395             }
396     }
397     proc guard_action_4 : sysClock {
398     if (sysClock == -1) return;
399             if (step == 1) {
400                     writeln(\"Guard raises his flag\");
401                     SendToEden(\"guard_flag = true; guard_raised_flag = true; step = 0;\n
    \");
402             }
403     }
404     proc guard_action_5 : sysClock {
405     if (sysClock == -1) return;
406             if (guard_flag && !isTrue(sm_flag)) {
407                     writeln(\"Guard lowers his flag\");
408                     SendToEden(\"guard_flag = false;\n\");
409             }
410     }
411     ");
412     }
413
414     proc delete_guard {
415     writeln("delete guard");
416     execute("
417     forget(\"guard_action_1\");
418     forget(\"guard_action_2\");
419     forget(\"guard_action_3\");
420     forget(\"guard_action_4\");
421     forget(\"guard_action_5\");
```

```
guard_raised_flag = @;
guard_flag = @;
step = @;
");
}

proc init_driver {
writeln("instantiating driver");
execute("
driver_ready = false;
proc driver_action_1 : sysClock {
if (sysClock == -1) return;
        if (isTrue(engaged) && isTrue(whistled) && !driver_ready) {
                writeln(\"Driver is ready to start engine\");
                SendToEden(\"driver_ready = true;\n\");
        }
}
proc driver_action_2 : sysClock {
if (sysClock == -1) return;
        if (isDefined(from) && isDefined(at) && isTrue(engaged) && from != at) {
                writeln(\"Changing destination of the train\");
                SendToEden(\"from = at; to = 3 - from;\n\");
        }
}
proc driver_action_3 : sysClock {
if (sysClock == -1) return;
        if (isTrue(engaged) && isTrue(can_move)) {
                writeln(\"Engine starts \");
                SendToEden(\"driver_ready = false; running = true;\n\");
        }
}
");
}

proc delete_driver {
writeln("delete driver");
execute("
forget(\"driver_action_1\");
forget(\"driver_action_2\");
forget(\"driver_action_3\");
driver_ready = @;
");
}

proc init_train {
writeln("instantiating train");
execute("
running = true;
brake = false;
door_open_1 = false;
door_open_2 = false;
from = 0;
to = 1;
at = 1;
engaging is running && to == at;
leaving is running && from == at;
engaged is !running;
alarm = false;
proc train_action_1 : sysClock {
if (sysClock == -1) return;
        if (engaging && !alarm) {
                writeln(\"Guard and station master finish their tea breaks\");
                SendToEden(\"init_guard(); init_sm(); alarm = true;\n\");
        }
}
proc train_action_2 : sysClock {
if (sysClock == -1) return;
        if (leaving && alarm) {
                writeln(\"Guard and station master off duty\");
                SendToEden(\"delete_guard(); delete_sm(); alarm = false; stop();\n\")
;
```



```
        }
}
proc train_action_3 : sysClock {
if (sysClock == -1) return;
        if (brake && running) {
                writeln(\"The train stops\");
                SendToEden(\"running = false;\n\");
        }
}
");
}

proc delete_train {
writeln("delete train");
execute("
forget(\"train_action_1\");
forget(\"train_action_2\");
forget(\"train_action_3\");
engaging = @;
leaving = @;
engaged = @;
running = @;
brake = @;
door_open_1 = @;
door_open_2 = @;
from = @;
to = @;
at = @;
alarm = @;
");
}

proc init_initialiser {
writeln("instantiating initialiser");
        Limit = 15;
}

proc SendToEden { para message;
        auto ok;
        ok = send_msg(stdmsg, [3, message], 0);
        if (ok == -1) error("can't write to message queue");
}

proc sleep { para period;
        auto start, current;
        start = ftime();
        for (current = ftime();
             (current[2] - start[2]) / 1000 + current[1] - start[1] < period;
             current = ftime());
}

stopClock = 1;
proc clocking : sysClock, stopClock {
        if (!stopClock && sysClock < stopTime) {
                sleep(0.5);
                if (sysClock != -1) {
                        SendToEden("sysClock = -1;\n");
                } else {
                        nextClock++;
                        SendToEden("writeln(\"time = \", nextClock);
                                sysClock = nextClock; \n");
                }
        }
}

/* iClock holds the positive system clock values, for external reference  */
proc syncClock : sysClock {
        if (sysClock != -1) iClock = sysClock;
}

proc startClk : startClock {
```

```
563             if (sysClock == @) nextClock = sysClock = 0;
564             stopClock = !startClock;
565     }
566
567     proc setClock { sysClock = nextClock = $1; }
568
569     srand(seed = time());
570     %srand(702605327);
571
572     init_door(1);
573     init_door(2);
574     d = Rand(2);
575     init_passenger(1,1,d,3-d);
576     d = Rand(2);
577     init_passenger(2,1,d,3-d);
578     d = Rand(2);
579     init_passenger(3,1,d,3-d);
580     init_driver();
581     init_initialiser();
582     init_train();
583
584     stopTime = 300;
585     startClock = 1;
```

The Railway Station Simulation Example

H.4. Extract of a Textual Simulation

```
instantiating door 1
instantiating door 2
instantiating passenger 1 1 2 1
instantiating passenger 2 1 2 1
instantiating passenger 3 1 1 2
instantiating driver
instantiating initialiser
instantiating train
Guard and station master finish their tea breaks
instantiating guard
Guard applies brake
instantiating sm
time = 1
The train stops
time = 2
Passenger 3 goes to the edge of the platform
Changing destination of the train
Guard is having a tea break
delete guard
time = 3
Passenger 3 opens door 1
time = 4
Passenger 3 is at the doorway
time = 5
Passenger 3 entering the train
time = 6
Passenger 3 closes door 1
time = 7
Passenger 3 takes a seat
time = 8
time = 9
time = 10
time = 11
time = 12
time = 13
time = 14
time = 15
time = 16
Station master whistles to call guard
instantiating guard
time = 17
Driver is ready to start engine
Station master stops whistling
Station master raises his flag
time = 18
Guard releases brake
time = 19
Guard raises his flag
time = 20
Station master lowers his flag
Train can move now
time = 21
Engine starts
Guard lowers his flag
time = 22
Guard and station master off duty
delete guard
delete sm
time = 23
time = 73
Guard and station master finish their tea breaks
instantiating guard
instantiating sm
time = 74
Guard applies brake
time = 75
The train stops
time = 76
Passenger 1 goes to the edge of the platform
Passenger 2 goes to the edge of the platform
Passenger 3 goes near door 1
```



```
Changing destination of the train
Guard is having a tea break
delete guard
time = 77
time = 78
Passenger 1 opens door 1
Passenger 3 opens door 1
time = 79
Passenger 2 is at the doorway
time = 80
Passenger 2 entering the train
time = 81
Passenger 2 takes a seat
time = 82
Passenger 1 is at the doorway
time = 83
Passenger 1 entering the train
time = 84
Passenger 3 is at the doorway
Passenger 1 takes a seat
time = 85
Passenger 3 alighting on platform
time = 86
Passenger 3 closes door 1
time = 87
Passenger 3 leaves the station
time = 88
delete passenger 3
instantiating passenger 3 2 2 1
time = 89
Passenger 3 goes to the edge of the platform
time = 90
Station master whistles to call guard
Passenger 3 opens door 2
instantiating guard
time = 91
Driver is ready to start engine
Station master stops whistling
time = 92
Passenger 3 is at the doorway
time = 93
Passenger 3 entering the train
time = 94
Passenger 3 closes door 2
time = 95
Station master raises his flag
Passenger 3 takes a seat
time = 96
Guard releases brake
time = 97
Guard raises his flag
time = 98
Station master lowers his flag
Train can move now
time = 99
Engine starts
Guard lowers his flag
time = 100
Guard and station master off duty
delete guard
delete sm
time = 101
time = 151
Guard and station master finish their tea breaks
instantiating guard
instantiating sm
time = 152
Guard applies brake
time = 153
The train stops
time = 154
```

```
143  Passenger 1 goes near door 1
144  Passenger 2 goes near door 1
145  Changing destination of the train
146  Passenger 3 goes near door 2
147  Guard is having a tea break
148  delete guard
149  time = 155
150  time = 156
151  time = 157
152  Passenger 1 opens door 1
153  time = 158
154  Passenger 2 is at the doorway
155  time = 159
156  Passenger 2 alighting on platform
157  time = 160
158  Passenger 2 leaves the station
159  time = 161
160  delete passenger 2
161  instantiating passenger 2 2 1 2
162  time = 162
163  Passenger 1 is at the doorway
164  Passenger 2 goes to the edge of the platform
165  time = 163
166  Passenger 1 alighting on platform
167  Passenger 2 opens door 2
168  time = 164
169  Passenger 3 is at the doorway
170  Passenger 1 closes door 1
171  time = 165
172  Passenger 1 leaves the station
173  Passenger 3 alighting on platform
174  time = 166
175  Passenger 3 leaves the station
176  delete passenger 1
177  instantiating passenger 1 1 2 1
178  time = 167
179  delete passenger 3
180  instantiating passenger 3 2 2 1
181  time = 168
182  Passenger 2 is at the doorway
183  time = 169
184  Passenger 2 entering the train
185  time = 170
186  Passenger 2 closes door 2
187  time = 171
188  Station master whistles to call guard
189  Passenger 2 takes a seat
190  instantiating guard
191  time = 172
192  Driver is ready to start engine
193  Station master stops whistling
194  Station master raises his flag
195  time = 173
196  Guard releases brake
197  time = 174
198  Guard raises his flag
199  time = 175
200  Station master lowers his flag
201  Train can move now
202  time = 176
203  Engine starts
204  Guard lowers his flag
205  time = 177
206  Guard and station master off duty
207  delete guard
208  delete sm
209  time = 178
210  time = 228
211  Guard and station master finish their tea breaks
212  instantiating guard
213  instantiating sm
```



```
214  time = 229
215  Guard applies brake
216  time = 230
217  The train stops
218  time = 231
219  Changing destination of the train
220  Passenger 2 goes near door 2
221  Passenger 1 goes to the edge of the platform
222  Passenger 3 goes to the edge of the platform
223  Guard is having a tea break
224  delete guard
225  time = 232
226  time = 233
227  Passenger 3 opens door 2
228  time = 234
229  Passenger 2 is at the doorway
230  time = 235
231  Passenger 2 alighting on platform
232  time = 236
233  Passenger 2 leaves the station
234  Passenger 1 opens door 1
235  time = 237
236  Passenger 3 is at the doorway
237  delete passenger 2
238  instantiating passenger 2 1 1 2
239  time = 238
240  Passenger 3 entering the train
241  time = 239
242  Passenger 1 is at the doorway
243  Passenger 3 closes door 2
244  time = 240
245  Passenger 1 entering the train
246  Passenger 3 takes a seat
247  time = 241
248  Passenger 1 closes door 1
249  time = 242
250  Passenger 1 takes a seat
251  time = 243
252  time = 244
253  time = 245
254  Station master whistles to call guard
255  instantiating guard
256  time = 246
257  Driver is ready to start engine
258  Station master stops whistling
259  Station master raises his flag
260  time = 247
261  Guard releases brake
262  time = 248
263  Guard raises his flag
264  time = 249
265  Station master lowers his flag
266  Train can move now
267  time = 250
268  Engine starts
269  Guard lowers his flag
270  time = 251
271  Guard and station master off duty
272  delete guard
273  delete sm
274  time = 252
275  time = 302
276  Guard and station master finish their tea breaks
277  instantiating guard
278  Guard applies brake
279  instantiating sm
```

The Railway Station Simulation Example

H.5. Scout Graphical Interface of the Simulation

```
%donald

viewport TEMPLATE

openshape Void
within Void {
        real dummy
        dummy = 0.0
}

openshape Body
within Body {
        line leftArm, rightArm, leftLeg, rightLeg, trunk
        leftArm = [(-30, -5), (0, 15)]
        rightArm = [(30, -5), (0, 15)]
        trunk = [(0, 15), (0, -10)]
        leftLeg = [(-35, -45), (0, -10)]
        rightLeg = [(35, -45), (0, -10)]
}

openshape Flag
within Flag {
        line L1, L2, Rod
        Rod = [(-30, -5), (-30, 40)]
        L1 = [(-30, 40), (-50, 20)]
        L2 = [(-50, 20), (-30, 20)]
}

openshape Whistle
within Whistle {
        line L
        circle C
        L = [(20, 40), (40, 40)]
        C = circle((40, 35), 5)
}

openshape Seat
within Seat {
        line L1, L2
        L1 = [(-40, -50), (40, -50)]
        L2 = [(40, 50), (40, -50)]
}

viewport RAIL

int distance                    # distance between stations
?_distance is distance;         /* bridging definition (B. D.) */

line edge, warningLine
edge = [(-1000, 0), (10000, 0)]
?A_warningLine = "linestyle=dashed,dash=12";
warningLine = [(-1000, -100), (10000, -100)]
label st1, st2
st1 = label("STATION 1", (distance + 400, -150))
st2 = label("STATION 2", (2 * distance + 400, -150))

openshape train
within train {
        int doorWidth
        line door1, door2
        point hinge1, hinge2, lock1, lock2, lock1ref, lock2ref
        line doorway11, doorway12, doorway21, doorway22
        shape seat1, seat2, seat3
        line shell1, shell2, shell3, shell4, shell5, shell6
        line interior1, interior2, interior3, interior4
        boolean door1open, door2open
        ?_train_door1open is isDefined(door_open_1) ? door_open_1 : 0;  /*B.D.*/
        ?_train_door2open is isDefined(door_open_2) ? door_open_2 : 0;  /*B.D.*/
        int at
        ?_train_at is isDefined(at) ? at : 0;   /*B.D.*/

        doorWidth = 80
        hinge1 = (100, 50) + (at * ~/distance, 0)
        lock1 = hinge1 + (doorWidth @ if door1open then -2 else 0)
        lock1ref = hinge1 + (doorWidth, 0)
        ?A_train_door1 = A_train_door2 = "linestyle=dashed,dash=42";
        door1 = [hinge1, lock1]
        hinge2 = (760, 50) + (at * ~/distance, 0)
        lock2 = hinge2 + (doorWidth @ if door2open then -2 else 0)
        lock2ref = hinge2 + (doorWidth, 0)
        door2 = [hinge2, lock2]

        doorway11 = [hinge1, hinge1 + (0, 25)]
        doorway12 = [lock1ref, lock1ref + (0, 25)]
        doorway21 = [hinge2, hinge2 + (0, 25)]
        doorway22 = [lock2ref, lock2ref + (0, 25)]
        shell1 = [lock1ref, hinge2]
        shell2 = [lock2ref, (940, 50) + (at * ~/distance, 0)]
        shell3 = [(940, 50) + (at * ~/distance, 0), \
                  (940, 450) + (at * ~/distance, 0)]
        shell4 = [(940, 450) + (at * ~/distance, 0), \
                  (0, 450) + (at * ~/distance, 0)]
        shell5 = [(0, 450) + (at * ~/distance, 0), \
                  (0,50) + (at * ~/distance, 0)]
        shell6 = [(0,50) + (at * ~/distance, 0), hinge1]
        interior1 = [lock1ref + (100, 0), lock1ref + (100, 150)]
        interior2 = [lock1ref + (100, 250), lock1ref + (100, 400)]
        interior3 = [hinge2 - (100, 0), hinge2 + (-100, 150)]
        interior4 = [hinge2 + (-100, 250), hinge2 + (-100, 400)]

        seat1 = trans(~/Seat, 370 + at * ~/distance, 250)
        seat2 = trans(~/Seat, 470 + at * ~/distance, 250)
        seat3 = trans(~/Seat, 570 + at * ~/distance, 250)

        boolean brake, running
        ?_train_brake is isTrue(brake);                 /*B.D.*/
        ?_train_running is isTrue(running);             /*B.D.*/
        label brakeStts, runningStts
        brakeStts = label( \
                if brake then "Brake applied" else "Brake released", \
                (at * ~/distance + 20, 400) \
        )
        runningStts = label( \
                if running then "Train running" else "Train on stop", \
                (at * ~/distance + 20, 350) \
        )
}

openshape P1
within P1 {
        shape body
        label head
        point position
        int at, pos, door
        ?_P1_at is isDefined(pat_1) ? pat_1 : 3;                /*B.D.*/
        ?_P1_pos is isDefined(pos_1) ? pos_1 : 3;               /*B.D.*/
        ?_P1_door is isDefined(door_1) ? door_1 : 1;            /*B.D.*/
        body = trans(~/Body, position.1, position.2)
        head = label("1", (-8, 20) + position)
        position = (at * ~/distance, 0) + \
                   (if door == 2 && pos != -2 then (660, 0) else (0, 0)) + \
                   (if pos == -2 then (370, 250) else \
                    if pos == -1 then (60, 150) else \
                    if pos == 0 then (140, 50) else \
                    if pos == 1 then (60, -50) else \
                    if pos == 2 then (60, -150) else (0, -300))
}

openshape P2
within P2 {
        shape body
        label head
```

```
143         point position
144         int at, pos, door
145         ?_P2_at is isDefined(pat_2) ? pat_2 : 3;                /*B.D.*/
146         ?_P2_pos is isDefined(pos_2) ? pos_2 : 3;               /*B.D.*/
147         ?_P2_door is isDefined(door_2) ? door_2 : 1;            /*B.D.*/
148         body = trans(~/Body, position.1, position.2)
149         head = label("2", (-8, 20) + position)
150         position = (at * ~/distance, 0) + \
151                    (if door == 2 && pos != -2 then (660, 0) else (0, 0)) + \
152                    (if pos == -2 then (470, 250) else \
153                     if pos == -1 then (140, 150) else \
154                     if pos == 0 then (140, 50) else \
155                     if pos == 1 then (140, -50) else \
156                     if pos == 2 then (140, -150) else (0, -300))
157 )
158
159 openshape P3
160 within P3 {
161         shape body
162         label head
163         point position
164         int at, pos, door
165         ?_P3_at is isDefined(pat_3) ? pat_3 : 3;                /*B.D.*/
166         ?_P3_pos is isDefined(pos_3) ? pos_3 : 3;               /*B.D.*/
167         ?_P3_door is isDefined(door_3) ? door_3 : 1;            /*B.D.*/
168         body = trans(~/Body, position.1, position.2)
169         head = label("3", (-8, 20) + position)
170         position = (at * ~/distance, 0) + \
171                    (if door == 2 && pos != -2 then (660, 0) else (0, 0)) + \
172                    (if pos == -2 then (570, 250) else \
173                     if pos == -1 then (220, 150) else \
174                     if pos == 0 then (140, 50) else \
175                     if pos == 1 then (220, -50) else \
176                     if pos == 2 then (220, -150) else (0, -300))
177 )
178
179 openshape SM
180 within SM {
181         shape body
182         label head
183         point pos
184         shape flag, whistle
185         boolean raiseFlag, blowWhistle
186         ?_SM_raiseFlag is isTrue(sm_flag);              /*B.D.*/
187         ?_SM_blowWhistle is isTrue(whistle);            /*B.D.*/
188         boolean rest
189         ?_SM_rest is !isDefined(sm_flag);       /* SM not exist */
190         body = trans(~/Body, pos.1, pos.2)
191         head = label("S", (-8, 20) + pos)
192         flag = trans(if raiseFlag then ~/Flag else ~/Void, pos.1, pos.2)
193         whistle = trans(if blowWhistle then ~/Whistle else ~/Void, pos.1, pos.2)
194         pos = (~/train/at * ~/distance, 0) + \
195               if rest then (480, -300) else (480, -50)
196 )
197
198 openshape Guard
199 within Guard {
200         shape body
201         label head
202         point pos
203         shape flag
204         boolean raiseFlag
205         ?_Guard_raiseFlag is isTrue(guard_flag);                /*B.D.*/
206         boolean rest
207         ?_Guard_rest is !isDefined(guard_flag);         /* guard not exist */
208         body = trans(~/Body, pos.1, pos.2)
209         head = label("G", (-8, 20) + pos)
210         flag = trans(if raiseFlag then ~/Flag else ~/Void, pos.1, pos.2)
211         pos = (~/train/at * ~/distance, 0) + \
212               if rest then (900, -300) else (900, 400)
213 )
```

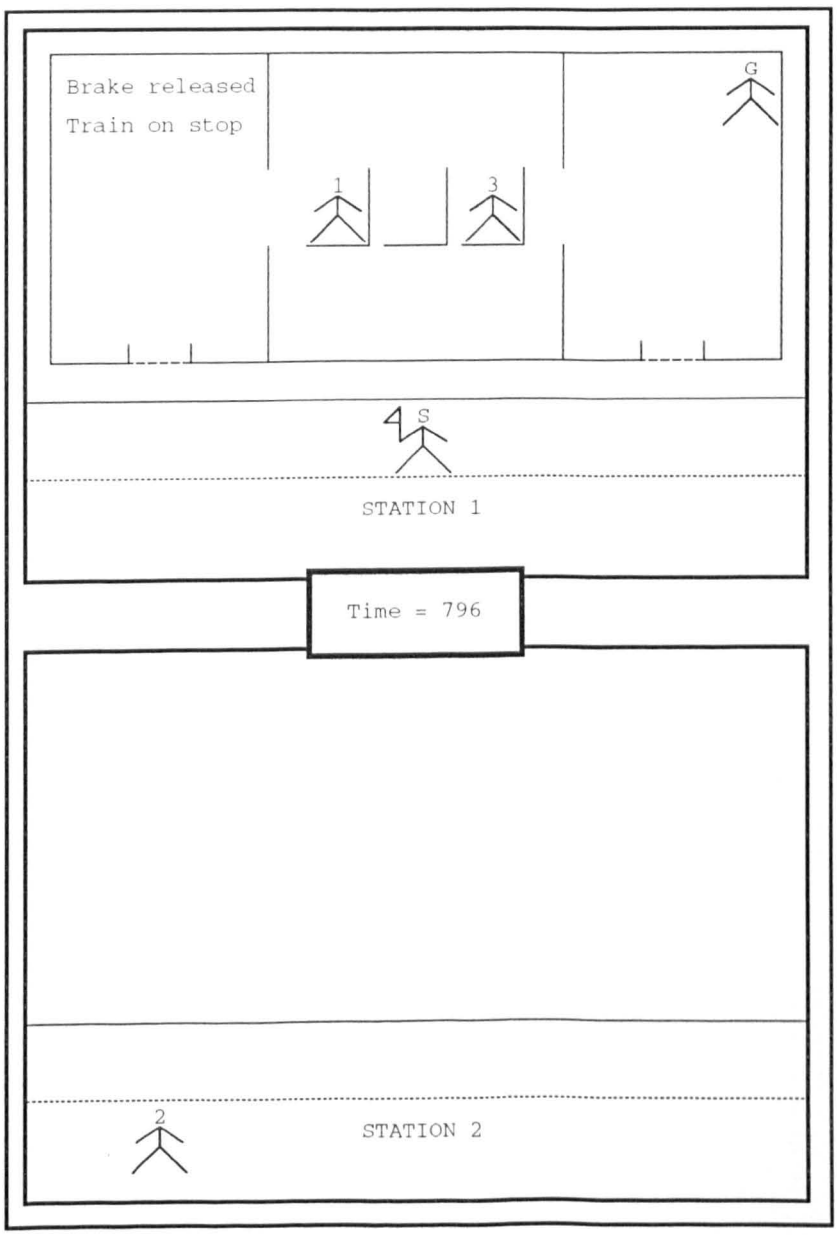


```
214
215 %scout
216 integer at, distance;   # EDEN variables
217 integer viewWidth, viewHeight, scale;
218
219 scale = 0.4;
220 viewWidth = 1000 * scale;
221 viewHeight = 710 * scale;
222
223 window station1, station2;
224
225 station1 = (
226         type: DONALD,
227         pict: "RAIL",
228         box: [(10, 10), (10 + viewWidth, 10 + viewHeight)],
229         xmin: 1 * distance -30,
230         xmax: 1 * distance + 970,
231         ymin: -230,
232         ymax: 480,
233         border: 2
234 );
235
236 station2 = (
237         type: DONALD,
238         pict: "RAIL",
239         box: [(10, viewHeight + 50), (10 + viewWidth, 50 + 2 * viewHeight)],
240         xmin: 2 * distance -30,
241         xmax: 2 * distance + 970,
242         ymin: -230,
243         ymax: 480,
244         border: 2
245 );
246
247 integer brake, running, iClock;
248 window clock, brakeStts, runningStts;
249
250 brakeStts = (
251         type:   TEXT,
252         frame:  ([(40, if (at==1) then 40 else viewHeight + 20 endif), 1, 20]),
253         string: if brake then "Brake is applied" else "Brake is released" endif,
254         alignment:      CENTRE,
255         border: 3
256 );
257
258 runningStts = (
259         type:   TEXT,
260         frame:  ([(40, if (at==1) then 60 else viewHeight + 60 endif), 1, 20]),
261         string: if brake then "Train is running" else "Train stopped" endif,
262         alignment:      CENTRE,
263         border: 2
264 );
265
266 clock = (
267         type:   TEXT,
268         frame:  ([(10 + viewWidth/2, 30 + viewHeight) - (7.5.c, 1.5.r), 3, 15]),
269         string: "\nTime = "//itos(iClock),
270         alignment:      CENTRE,
271         border: 3
272 );
273
274 screen = <clock / station1 / station2>;
275
276 %eden
277 /* testing data */
278 iClock = 0;
279 at = 1;
280 distance = 1000;
281 door_open_1 = 1;
282 door_open_2 = 0;
283 pat_1 = 1;
284 pat_2 = 2;
```

```
285   pat_3 = 1;
286   pos_1 = -1;
287   pos_2 = -2;
288   pos_3 = 0;
289   door_1 = 1;
290   door_2 = 2;
291   door_3 = 1;
292   /* end of testing data */
```

The Railway Station Simulation Example

H.6. A Sample of the Graphical Output


Brake released
Train on stop
G
1
3
S
STATION 1
Time = 796
2
STATION 2