A DISTRIBUTED REAL-TIME LANGUAGE AND ITS OPERATIONAL SEMANTICS

by

Padmanabhan Krishnan

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1989

To my parents

ACKNOWLEDGEMENTS

My deepest gratitude is to my advisor Professor Richard Volz for his constant, untiring encouragement and support throughout the course of this work. It is impossible to imagine a person who would be more concerned about his students and this work owes a great deal to his devoting many hours of his time in discussing various aspects of this work.

I would also like to thank my co-chair, Profesor Trevor Mudge for his critical evaluation and guidance. I am also grateful to other members of my advisory committee, Professors Andreas Blass, Chinya Ravishankar, John Sayler and Toby Teorey for their critiques and guidance. Thanks also to Professors Larry Moss and Satish Thatte for having taken a special interest in my work.

Thanks to Charles Antonelli, Venu Banda, Jarir Chaar, Russell Clapp, Dilip Kandlur, Oyekunle Olukotun, Sriram Padmanabhan, Ravi Rao, Fritz Ruehr and Suresh Subramanian for making the sometimes tedious process of research enjoyable. This research has its roots in the distributed systems seminar organized by the Robot System Division and I am thankful to them for the discussions on various topics. I would also like to thank the staff at the EECS department at the University of Michigan and at the Computer Science department at Texas A&M University.

Finally, I am grateful to my parents and sisters for encouraging me to pursue a higher education.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Motivation

With the proliferation of inexpensive computing power, distributed computing — computing in parallel across a number of machines — has become very popular. Despite a plethora of distributed systems, there is very little available by the way of well defined programming paradigms, or formal specification tools for these systems. A similar scenario exists in the world of real-time systems. A distinguishing property of real-time systems is the presence of timing constraints. That is, it is not just sufficient to compute the required function but it is essential to compute the necessary function within certain time bounds. Even though a large number of real-time systems have been constructed, there is no single language/model which supports techniques to express the various constraints of a real-time system, analyze them for consistency and implement the system.

The techniques to develop distributed real-time systems are very ad-hoc. Because of this, such systems are expensive to build and maintain. The skyrocketing costs of distributed real-time systems are of major concern, as these systems are prevalent in many application domains like avionic systems and manufacturing systems, and each system costs an astronomical sum per year [12].

The reason for this ad-hoc nature is that one does not have a usable and yet sufficiently abstract computation model of distributed real-time systems. If one had a model of computation, one could then build a compendium of information based on this model which will help an implementor avoid re-solving problems that have been solved elsewhere. Therefore a model of computation would ease the building of software systems. This work is the first step in constructing such a model of distributed real-time computation.

A valid question the reader might raise is why we choose to model distributed real-time systems when we have stated that the individual components of distributed systems and real-time systems have not yet been well understood. The reason for this stems from our belief that all systems can be classified as one of the following: *sequential*, *concurrent*, *concurrent real-time*, *distributed* or *distributed real-time*. Notice that in the above classification, we do not distinguish between distributed and parallel computation. Generally speaking, parallel computation refers to computing in an environment with a 'large number' of processes executing a similar code. A parallel system can be treated as a distributed system, with a large number of computing elements.

A natural containment relation between programs for these various systems exists. For example, sequential programs are a special case of concurrent programs. If these systems are arranged in increasing order of programming and semantic (and hence comprehension) complexity, they form a partial order as shown in figure 1.1. If one is able to design a model to denote the top or the most complex element in the partial order, the same model can be used to express all elements below it.

It is also the case that more and more real-time systems are being developed for distributed execution. This is because of the reduction in overall execution time in the case of distributed systems. Therefore it is only natural that we model distributed real-time systems.

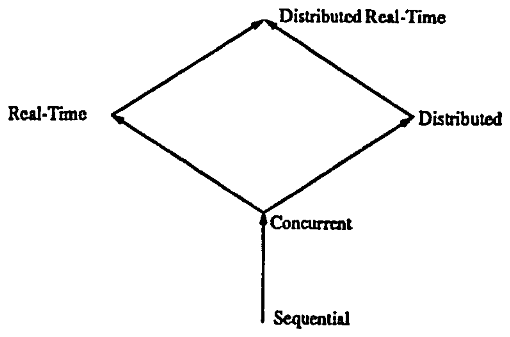


Figure 1.1. Ordering of System Complexity

1.2 Goals

In [81], Stankovic enumerates the challenges presented by real-time systems. The problems are categorized into specification and verification, real-time scheduling theory, real-time operating systems, real-time programming languages and design methodology, distributed real-time databases, artificial intelligence, fault tolerance, real-time system architectures and real-time communication. In this thesis a subset of these issues is addressed. The scope is limited to programming languages for specification of distributed real-time systems.

The principal issue in specification of real-time systems is how to deal with time. [51] defines a number of standards of time. When choosing a standard (or a set of standards), one has to be careful not to cripple the number of systems that can be specified. The choice of an appropriate standard is discussed in chapter 4. Irrespective of the chosen standard, there are two principal problems in including time explicitly. The first is that it presents problems in automatic verification. The second problem arises due of a multiple notion of time (one per site). In such a situation there is a need to address the issues of clock synchronization and creating a universal time frame in the system. However, as discussed in a later section the notion of time cannot be done away with.

We assume that languages can be used to characterize models of computation just as regular languages model finite automata. This then permits the building of a computation model using a programming language. We have designed a programming language, as against only a formal language, as we wish the model to be useful in practice and aid in system building. We emphasize that we do wish to construct a formal model also. Hence the first step was to define a formal semantics for the language. Other characteristics necessary for a model (like proof theory etc.) must be derived from the defined semantics.

To model real-time systems correctly, all timing constraints of the hardware on which the program is to actually execute have to be modeled. One also needs to measure the time necessary to execute each programming construct. However, we do not wish to burden ourselves by considering the execution speed of programs written in the language at this stage. As our main concern is that of expressibility of the timing constraints, the language will be used primarily for rapid prototyping. That is, the language will facilitate specifying and building a prototype of the system quickly. Benchmarking of implementations will give us information of which constructs are expensive to use. This information will be useful when production quality systems are built. But this is too removed from our immediate goal and shall not be discussed further.

Also performance related issues are tied to specific architectures and specific implementations. A system specifier should be able to design a system which is independent of architecture and be able to map it onto any given architecture. As mentioned in [17], the first concern is to design a solution to the problem, while efficient implementation of it on a particular architecture is only relevant after the design. In order to have a realistic real-time environment for the execution of a prototype, time has to be scaled appropriately to account for delays introduced by the implementation of the language.

In short, the basic goal of this research is to design a simple, but also sufficiently expressive computational model to permit the specification of distributed-real-time systems. This goal can be divided into two major sub-goals. The first is to design a programming language to be used for prototyping distributed real-time systems. The second is to take the first step in developing the theoretical underpinnings of distributed real-time computation by developing the semantics for the language.

1.3 Approach

In this section the strategy for the development of the language for distributed real-time systems, and the building of its corresponding model is discussed. The first step is the selection of an appropriate programming paradigm after which one has to develop a notion of time and distribution within the chosen paradigm. We also identify suitable constructs for specification of the temporal constraints and incorporate them into the language. Having done that we define the semantics for the language. In the following sections, our choice of programming paradigm, time, distribution, temporal specifications and semantics are introduced.

1.3.1 Paradigm

In the construction of a programming language, the *programming paradigm* is of paramount importance. The three major programming paradigms currently in vogue are *imperative*, *functional* or *applicative* and *logic*.

The *imperative* style includes languages like C [49] and Ada [1]. The key feature of the imperative style is the concept of a mutable state. These languages usually support several control structures (loops, if-then-else, case) and are based on the assignment statement which changes the state of the computation [8]. *Functional* languages (like Miranda [1] [88]) are stateless and allow only single assignment to variables. That is, the values associated with variables cannot be altered and are more like constants. As the main emphasis is on function composition and function application, functional languages support program building rather than object building [7]. Programming in *Logic* is best explained by describing Prolog [53]. Prolog is based on a subset of first order logic known as Horn clauses. A set of Horn clauses defines the problem domain. The particular problem to be solved is also

[1] Miranda is a trade mark of Research Software Ltd.

posed as a set of Horn clauses which are to be proven in the original domain. The solving of the problem is similar to finding a proof for the given set of clauses given the original set of clauses which are to be treated as axioms. How the proof is to be achieved, or the control information, is not specified by the programmer. The language implementor makes the decision of how to effect the proof procedure. See [83] for details.

We choose a paradigm which is essentially imperative. However, certain sub-sets of programs in the language can be identified as functional/declarative. The driving force behind the decision opting for an imperative paradigm was the need to support a notion of time in the language. Advantages of functional programming are obtained by the identification of functional sub-sets. The details of this are presented are chapter 2. The declarative component of the language was necessary to characterize the environment in which the system operates.

1.3.2 Concept of Time

Usually, it is difficult to specify or program a real-time system in a language that does not support any metric which can be used to measure the rate of progress of the computation or presence of events in the environment. A commonly used metric is time. We, as real-time language designers, decided to have the notion of *time* in the language. Two techniques can be used to achieve this. One can either have explicit clocks or use events to keep track of time. Explicit clocks represent time as a monotonically increasing function (except for clock synchronization). Events on the other hand express intervals of time. If it is known that event b occurs t units of time after a, event a can reset time to 0 and when b has occurred we know that t units of time has elapsed. The latter is not very useful in our opinion, as it is extremely difficult to specify event occurrences with much precision. Usually, only the upper and lower bound of the interval is specified. In such a situation the sense of time is not very accurate. Our language utilizes explicit clock definition to support the notion of time.

1.3.3 Denotation of Distribution

One of the goals of the research is to support the specification of distributed systems. The language must have construct(s) which can be interpreted to represent distribution as opposed to concurrency. No programming language we surveyed (see chapter 2) distinguishes concurrent programming and distributed programming.

Therefore, at the outset of this research, we identified certain features which could be used to distinguish concurrent and distributed computing. These features are orthogonal in that selection of one as a denotation of distribution does not diminish the need to support the others. We enumerate these features and discuss their implications.

In a distributed system, a particular value of interest to a computation may not reside on the site of computation. In such a case a *remote access* is essential. For example, in a distributed data base, all the items may not be replicated, to avoid the update overhead, consistency problems etc. Hence a query may need to perform a remote access.

An important fact which could be used in the design of algorithms for distributed systems is that access to a remote variable could take orders of magnitude more time than a local access [95]. This important difference can be used to characterize a distributed system. Obviously one cannot describe the difference quantitatively as it is extremely difficult to account for network delays precisely. For remote access to be a distinguishing feature for distributed system, constructs denoting remoteness using qualitative descriptions are necessary.

The *remote access* technique is useful only if one allows sharing of data. On the other hand if sharing of data is disallowed (as in CSP [44]), other techniques to characterize a distributed system are essential. Some of the other alternatives are discussed below.

In a uniprocessor system, an occurrence of a fault results in an abnormal termination of the entire program. However, a distributed system can be designed to continue to perform at least a subset of the functions at a lower throughput in the event of a fault. This can be achieved by reconfiguring the system to perform the desired functions. If one had constructs which specify how a system should *reconfigure* itself in the event of a fault, those constructs could be interpreted as a denotation for a distributed system. However, this would imply that our language will have to provide specific techniques for fault tolerance. Having a language dictate fault handling schemes is far from ideal, as different types of faults require different schemes. The fault handling scheme will also depend on what the program is computing and when the fault occurs. A language should provide general constructs which can be used for all types of recovery without requiring any particular technique be used. Hence reconfiguration specification techniques were not included in the study.

It is reasonable to assume that each processor in a distributed system has a *single clock* associated with it. Under the above assumption, one can require a specification of a distributed system to involve multiple clocks. Note that this does not require the clocks to be completely independent. The various clocks could be synchronized with each other to keep a uniform notion of time with an acceptable margin of error. However, a model must be sensitive to clock synchronization and other errors introduced to a multi-time notion.

For the purposes of this thesis, we select one of the above as the *main feature denoting distribution.* This is not to conclude that the other features cannot be used to characterize distribution nor do we conclude that a distributed system consists *only* of multiple clocks. It is our opinion that multiple clocks are an important component of distribution and we concentrate on the issues related to multiple clocks.

The main criterion in the selection was that the chosen characterization should be compatible with the rest of the language. Another important property that the selected characterization should have is that programs using it should not be overly verbose. Out of the three possible denotations discussed above (remote access, reconfiguration and multiple clocks), the multiple clock characterization was considered, as it introduces a sense of time in the model and at the same time described a distributed system. Also this technique is general enough to be added to existing languages.

1.3.4 Temporal Specification

In a programming language for real-time systems, it is essential to support techniques which specify timing constraints the system is required to satisfy. As we have an interest in developing a formal model, we assess the suitability of temporal logic [57] for the above and overcome its shortcomings to suit our purpose. As discussed in the next chapter, classical temporal logic is insufficient as far as real-time systems are concerned. In this thesis, we discuss certain extensions to temporal logic.

1.3.5 Formal Semantics and Model

Having decided upon the basic structure of the language, it is essential to have a formal framework within which one can analyze and reason about programs. Before analysis, the meaning of syntactic terms has to be clear. The definition of formal semantics provides meaning to the syntactic constructs of the language. From the semantics, the tools for analysis are to be constructed. Hence, the definition of semantics is the first step towards the goal of buiding a model. Semantics are also necessary to make sure that all implementations do implement the same language.

The purpose of semantics is to assign a meaning to a program, which describes the effect of executing the program. This is useful to a system designer who is interested in knowing what happens when a particular construct is executed. Certain types of semantics also provide a formal basis for deducing properties of a program. Mathematically speaking, a syntactically correct program P can be considered to be an element of a set Π of all syntactically correct programs. The meaning of a program P can be then be denoted by $\mathcal{M}(P)$. To define a formal semantics one has to specify a mapping from Π to $\mathcal{M}(\Pi)$. To define this mapping completely the structure of the elements of $\mathcal{M}(\Pi)$ has to be properly chosen. These elements can be considered to be the elements of the model of distributed real-time computation.

There are two main semantic styles: *denotational* and *operational.* Denotational semantics while founded upon an elegant and powerful theory has problems mainly because it is not easy to understand. As pointed out in [66] and [71], this is partly because of the use of higher order functions taken as denotations of phrases. This and other shortcomings of denotational semantics shall be discussed in chapter 3.

It is widely accepted that operational semantics is easier to understand than denotational semantics. Operational semantics can be considered to be an abstract implementation. It is usually associated with an abstract machine which executes programs in the language. Initial attempts at operational semantics [99, 59] to cite a few, (see chapter 3 for a detailed survey) resulted in an overly verbose characterization and tended to be biased towards specific implementation strategies. Structural Operational Semantics [72] used axiomatic descriptions of transitions and overcame some of the problems in traditional operational semantics.

The advantage of operational semantics over denotational is that it is more understandable as the reader is able to imagine the execution of the program thus visualizing behavior. In other words, if the purpose of semantics is to explain the behavior of programs written in the language then an operational semantics is necessary. This is because most people understand programs by visualizing their behavior. An operational or a behavioral view of the language captures meaning by formalizing the idea of behavior and explaining the effect of executing each syntactic entity. It is for this reason that in this thesis, we develop an operational semantics for the language.

Summary

In this thesis, we develop A Real-time Language called ARL. ARL derives its name from APL. Just as APL is an expressively powerful language for sequential computation, ARL is powerful for distributed real-time computation.

ARL can be considered to be an imperative extension of a functional language. It uses explicitly defined logical clocks to represent time. Multiple such definitions will characterize a distributed system. It supports the definition of temporal constraints. We have also defined an operational semantics for ARL.

As a brief overview, this thesis addresses the issues related to selection of programming paradigm, characterization of time and distribution, constructing a formal and programming model of distributed real-time systems. In the next chapter, we review very briefly some of the issues related to programming languages and distributed real-time languages in particular. In the same chapter we explain in detail our decision to have an imperative language with functional sub-set. The relation between the programming paradigm and clocks as time keepers, events as communications and temporal specifications is also discussed. In the third chapter we review some of the relevant semantic techniques. We also discuss concurrent models of computation which can be used as semantic models. The syntactic structure of the language is explained in chapter four, while the formal semantics are described in chapter five. In chapter six we demonstrate using examples the advantages of our language over other languages. Finally in chapter seven we summarize our achievements and identify areas which require future research.

CHAPTER 2

REVIEW OF PROGRAMMING LANGUAGES

In this chapter, issues related to the design of programming languages are discussed. The main emphasis is on the identification of programming language principles and study of their strengths and limitations for distributed/real-time systems. The issues discussed include the three principal programming paradigms and their amalgamation, the concept of events for communication and asynchronous behavior, and temporal logic for timing requirements. Finally, a few languages designed explicitly for distributed/real-time systems are studied.

2.1 Programming Paradigms

As mentioned in chapter 1, three principal paradigms of programming namely, functional, logic and imperative are considered. After studying their suitability for prototyping distributed and real-time systems, features that are relevant to our goal are identified.

2.1.1 Functional Programming

There is a growing interest in functional languages and systems and a number of languages have been designed, FP [7], SASL [90], Miranda [88] to cite a few. There are both theoretical and practical advantages in using the functional paradigm. The theoretical advantages of functional languages over assignment based or imperative languages lies in the fact that lambda calculus is a natural model for their mathematical semantics. From a user's point of view, functional languages are useful as it is easy to support type inferencing, polymorphism, pattern matching, lazy evaluation, higher order functions, etc. These features enable one to arrive at an executable specification for a problem quickly as demonstrated in [86] and [87].

Algorithms can be considered to be composed of *logic* and *control* [52]. The logic part specifies what the algorithm achieves, i.e. *what* is being computed, while the control part denotes *how* some function is to be achieved without explicitly telling what the function computes. The main emphasis in a functional language is on the logic along with some information about the control of the solution rather than purely control or purely logic.

2.1.1.1 Limitations of Functional Languages

Theoretically, expressions which are purely functional can be executed in parallel. This fact can be used to speed up the execution of a functional program. However, from a pragmatic view point they execute slowly on current hardware. This restricts the usage of functional language in production quality systems. Languages which support normal order evaluation (e.g., lazy languages) are difficult to debug. Standard debugging techniques are stack based and cannot be applied to them. This is because a graph reduction scheme [91] is used to implement normal order evaluation.

From a logical expressiveness view point functional languages are well suited for sequential programs. However, they are not adequate for concurrent, distributed, real-time program specification. A sequential system can be modeled by functions modifying values computed by other functions. Usually concurrency involves non-determinism and synchronization. Non-determinism cannot be modeled using a purely functional approach as a non-deterministic function can produce different outputs given the same input and hence is not functional in the mathematical sense.

For example, consider a merge routine, which given two input streams merges them into one stream. Assume that the input streams are to be considered equivalent. Also assume that an implementation of merge routine should be permitted to non-deterministically select data from either stream when data is available on both. Thus merge([1][2]) should be allowed to produce either [1,2] or [2,1] and hence cannot be a function. Note that an implementation of merge is bound to be deterministic but the specification of merge at the language level should be non-deterministic.

Synchronization is more of a state characteristic than a value characteristic and cannot be represented by values returned by functions. Concurrent programs have to be specified in terms of the behavior or collection of properties rather than only values. Also, time being an implicit side-effect cannot be specified in a purely functional language.

2.1.2 Imperative Programming

Concurrent/real-time imperative languages [1, 10], do not have a natural formal model associated with them as applicative languages have lambda calculus. But they are very attractive for production quality software because of their wide spread use and execution efficiency on current architectures.

However, they lack the expressiveness to support rapid prototyping. This is especially true if an imperative programmer has to have a type secure program. Typing plays an important role in program debugging. As debugging is an important activity when building a prototype, the type structure provided by the language is important. In imperative languages, the onus of providing types for the objects and functions is on the programmer thereby resulting in verbose code. This is not acceptable for prototyping system. It is also the case that the builder is unnecessarily burdened with efficiency considerations from the start and the specificational aspects get side-lined [17].

Because these languages support the concept of mutable state, the specification of time and synchronization in these languages is relatively simple.

2.1.3 Logic Programming

As logic programming's primary concern is with the logic of the system being programmed, it has a distinct advantage in the area of specificational languages. This is because specifications should be precise and yet allow the implementor the freedom to choose the method of implementation, i.e., they should contain only the logic of the system. Even though logic programming appears to be the most appropriate, it has some shortcomings when one considers distributed/concurrent logic programming.

Two types of parallelisms have been considered in logic programming, *explicit concurrency* and *implicit concurrency* [62]. Implicit parallelism is concerned with the definition of parallel implementations of the language that are equivalent to the sequential semantics of the language. Explicit concurrency is obtained by introducing constructs related to concurrency. We are concerned with explicit concurrency as our aim is to describe parallel systems, where the specifier may wish to specify what can be computed in parallel.

Three basic mechanisms to transform a sequential logic language to a concurrent language have been proposed. *Synchronization operators* in Generalized Horn Clauses [27] are similar to process fork and join operators. This makes the logic complex and hard to understand. *Synchronous communication mechanisms* are similar to those of CSP. The mechanisms are based on the notion of events. *Constraints* on the logical variables are present in languages like Parlog [18] and Concurrent Prolog [79]. These languages require annotations of variables as read only to achieve synchronization. It is extremely difficult to control the proof tree and these constructs, when added to other constructs like *cut*, only make the program more difficult to understand.

In general, logic languages are less expressive than functional languages for concurrent systems, because logic languages have no control information. The notion of synchronization is not only state based, but also requires control information. Due to this, logic programming does not play a major role in our language. However, our language (ARL) does support certain constructs which are declarative in nature.

Conclusion

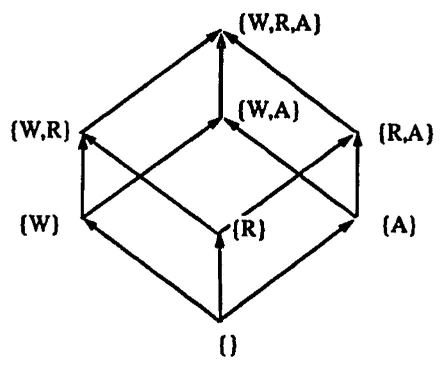
Based on the above arguments, we conclude that, while functional programs specify the logic more explicitly than imperative languages, they lack the notion of state and the various control structures available in the latter. Imperative languages on the other hand specify the control associated with the algorithm in detail and it is difficult to extract the logic from them. As executable specifications should embody the logic explicitly along with some control information, functional languages can be considered as a better vehicle for rapid prototyping than conventional or logic languages.

However, not all tasks, especially in distributed/real-time computation, can be done within the functional paradigm. To adapt a quote from Turner [89], "... certain aspects of the language must be imperative and any attempt to pretend otherwise can only be an exercise in self deception." Thus the applicative paradigm is combined with the imperative paradigm to form the basis for our language. This makes the language essentially imperative, but certain parts of it can be identified as functional.

2.2 Combining Functional and Imperative Languages

Having decided to combine the two paradigms, the possibility of retaining the best of both worlds in this mixed paradigm is discussed. Gifford et al [31] present a method by which it is possible to write functional and imperative computations in the same program. The principal property which distinguishes functional programming is that *all expressions* in a functional program are *referentially transparent*.

A sub-expression in an expression is said to be *referentially transparent* if it can be replaced by another expression having the same value without any effect on the value of the whole expression. This property can be invalidated if the expressions use any of the following three operations: 1) The ability to *allocate* and initialize memory whose values might change, 2) The ability to *read* the contents of memory whose value might change, 3) The ability to *write* new values into existing memory locations.



R: Read variables
W: Modify variables
A: Allocate memory

Figure 2.1. Effect Class

A given expression might use a subset of the three operators described above. To characterize this, we consider the power set of the set of operations. It has eight elements and each of these eight subsets is called an *effect class*. A complete partial ordering based on the subset relation of the power set is given in figure 2.1. The complete partial order can be reduced to form other partial orders depending on the type of restrictions one places on the system. [31] discusses the various restrictions in detail. As our main concern is to be able to distinguish functional from non-functional program fragments, we collapse the lattice into a linear order with three elements called FUNCTIONS, OBSERVERS and PROCEDURES. FUNCTIONS are purely applicative, while OBSERVER can see side effects, i.e., access state variables but not change them, and PROCEDURES can cause and observe side effects.

Inference rules are used to decide the effect class of a subprogram. These inference rules partition a program into effect classes. For example, the fact that constants are functions and variables are observers can be stated as inference rules.

To do so, define A, an extended type assignment and two functions *Type* and *Effect*. An extended type assignment is a partial function that maps an identifier into a pair consisting of its type and a token 'Constant' or 'Variable'. The function *Type* maps an extended type assignment and an expression into a type, while the function *Effect* maps an extended type assignment and an expression or a statement into an effect class. The inference rules have two components to them. The first is the antecedent, while the second is the consequent and a horizontal line separates them. Examples demonstrating how inference rules can be defined are given below.

$$\frac{A(\mathrm{v}) = <\mathrm{T,Constant}>}{\mathit{Effect}(A)(\mathrm{v}) = \mathrm{Function}}$$

$$\frac{A(\mathrm{v}) = <\mathrm{T,Variable}>}{\mathit{Effect}(A)(\mathrm{v}) = \mathrm{Observer}}$$

The first of the two rules described above classifies a constant 'v' as a Function, while a variable 'v' is classified as an Observer. The rule described below classifies the assignment operator as a procedure. In it E represents an expression, and v a variable of type T. The rules states that if v is a variable of type T and the type of E is contained in T, the effect class of 'v := E' is procedure.

$$\frac{\begin{array}{c} A(\mathrm{v}) = <\mathrm{T,Variable}> \\ \mathit{Type}(A)(\mathrm{E}) \subseteq \mathrm{T} \end{array}}{\mathit{Effect}(A)(\mathrm{v} := \mathrm{E}) = \mathrm{Procedure}}$$

However, this classification is necessarily conservative in that a subprogram will be assumed to be a procedure unless proven otherwise. This is because it is undecidable whether a subprogram actually destroys referential transparency.

We have similar inference rules designed specifically for ARL to classify the program into the various classes. It is now possible to use appropriate techniques to implement and formally describe elements in each class. Therefore one can write a program with both imperative and functional constructs and classify them using the described scheme.

2.3 Events

Events are used in various programming languages but there is no single interpretation of what for and how they should be used. Their primary use has been to model information transfer in a concurrent environment. We discuss a few of the event oriented models. This will enable us to identify some of the issues in supporting events at the language level. We discuss Actors [42, 2], Group Element Method (GEM) [60], Event Based Language (EBL) [77] as examples of event based models.

Actors

In an Actor model, the basic elements are actors and events. An actor is a computational agent which maps each incoming communication to a 3-tuple consisting of 1) A finite set of communications sent to other actors 2) A new behavior which governs the action to the next communication. 3) A finite set of new actors created by the communication.

The three elements can be interpreted as follows. An actor can be considered to be a process which handles messages. The process could contain local information retained from event to event (like a history.) A process also has the ability to spawn other processes. An event is said to occur when a message is received by an actor. An event signifies the start of an action following a communication. The complete theory related to Actors is developed in [2] and is not discussed here. The following is an example of an Actor based factorial function which illustrates the use of events.

```
def expr fact [n] =
            if n = 0 then
                reply[1]
            else
                reply [n * (call self (n-1) )]
```

In the above example, reply denotes sending the response to the actor that invoked fact, while call self is a recursive call.

Group Element Method

An event in the Group Element Method(GEM) [60] represents a logical atomic action marking the end of an action. A computation is represented by a concurrent execution of events related by partial orders. We concentrate on the notion of events and do not discuss the computational aspects.

An event instance belongs to a class called an event type. For example, Assign(newval : Integer) defines an event type called Assign. All instances of it have an integer input parameter. Thus put(5) is a legal instance of Assign where put is the name of the specific event and 5 the argument.

In general, an event contains data and structural information. The data field consists of a set of parameter values and what are called thread identifiers. Parameters are the normal static data items like integers, boolean etc. Threads on the other hand identify specific control paths. The structural information consists of a name and two identifiers called the element and group identifiers.

An element can be thought of as a sequential process while a group can be considered to be an encapsulating unit (e.g., a module.) A thread is an identifier for a particular chain of events. For example, (e,(p1,p2,...pn),t,elem,grp) identifies an event of type e with value (p1,p2,...pn) in thread t. The event instance belongs to element e in group grp.

Temporal logic is used to specify restrictions on event generation. The specification of various systems using GEM is discussed in detail in [60].

Event Based Language

[77] discusses an Event Based Language(EBL) where events are abstract entities that are explicitly instantiated by a program during its course of execution. A program is composed of processes which can be executed concurrently.

An event instance carries with it information about its occurrence. Two relations involving events, named *causality* and *precedes*, are defined. The causality relation is used to indicate the process which causes an event and the precedes relation indicates the ordering of events. In EBL, events are used for all programming activities including assignment. Therefore events are not necessarily atomic.

Discussion

The common feature in the use of events as discussed above, is that events represent *significant points* in the computation. Statements to generate events explicitly are provided. They can be used to represent explicitly used entities ranging from assignment to communication.

The notion of events can also be extended to model entities not explicitly instantiated but handled by the program like external interrupts. In general, events can be used to represent interrupts, I/O, exceptions etc. Events can also be used to denote either atomic or non-atomic activities. In short, the use of events has been to indicate a change of state of the computation.

If events are to be useful, the information associated with them should be state based. Therefore they cannot be a part of the functional paradigm. As real-time programming requires a notion of state, ARL supports the definition and use of events.

Events also play an important role in modeling communication in a distributed environment. It is possible to characterize asynchrony using events. However, if asynchronous event generation and handling has to be reliable, the implementation has to handle unbounded buffering.

In the next section, we examine the usefulness of temporal logic for real-time computation. Discussed are two principal types of temporal logic viz., linear time logic and interval logic. We do not discuss other temporal logics like branching time logic etc. [25].

2.4 Temporal Logic

A logical framework is essential to facilitate specification and analysis of real-time constraints. Temporal logic is a candidate formalism, as it is a logic dealing with a notion of time. Temporal logic is essentially state based and it enables one to write predicates about a particular state or a sequence of states as they are reached during the course of the computation. As shown by Lamport [58], a program can be specified by a set of properties it satisfies.

The basic operators in temporal logic are $\Box$ representing henceforth, $\Diamond$ indicating eventually and $\circ$ indicating next. Towards their meaning, let p be a predicate. $\Box$ p implies that p is true from now on till the computation terminates. $\Diamond$ p means that p is true now or will become true sometime in the future. $\circ$ p implies that the predicate will be true at the next state of the computation. A few definitions used in the context of temporal logic are presented.

Definition 2.1 A predicate p is a safety property if it is initially true and holds at every state of the computation, i.e. p is invariant throughout the computation.

Intuitively a safety property states that no incorrect state is ever reached during the entire computation. For example: $\Box p$ is a safety property. As all safety properties are satisfied by a null program — a program which does nothing — we also require *liveness* properties which require that something good actually happens.

Definition 2.2 A predicate p is a *liveness* property if p is true at some state(s) of the computation.

The following are examples of liveness properties: $\Diamond p$, $\Box\Diamond p$, $\Diamond\Box p$.

Definition 2.3 A non-terminating process is usually called a *Divergent* process.

Temporal logic has to been used mainly to state and/or prove safety and liveness properties in correctness of programs. It has been successfully used for reasoning and verification about concurrent programs [68, 39]. Our interest in temporal logic is different from the above. We are concerned with the appropriateness of temporal logic as a vehicle to express timing constraints of a real-time system. A stricter notion of liveness called timeliness is necessary. Timeliness (which is defined below) has two components to it. The first concern is that a predicate becomes true only after at least n units of time, while the second concern is that a predicate becomes true within n units of time.

Definition 2.4 The at least case is defined as follows: $\neg p$ & $\forall i \leq n\ o_i \neg p$ & $\exists m > n\ o_m p$.

Definition 2.5 The within case is defined as follows: $\neg p$ & $\exists i \leq n\ o_i p$.

Many real-time applications are required to be divergent and hence one should be able to specify divergent computation. We, would however, like to distinguish two types of divergences namely, internal divergence and external divergence. Internal divergence occurs when the system diverges without any further interaction with the environment, while external divergence occurs when the system diverges but maintains a continuous interaction with the environment. Ideally, real-time systems like control systems should be externally divergent, but generally no system should be internally divergent. A requirement of the logic we wish to use is that one should be able to specify external divergence. It would be an added advantage if it were possible to detect certain types of internal divergence in a specification.

Interval logic [4], which is a variation of the temporal logic discussed above, can be used to represent a finer granularity of time than the operators described above. The following operators are defined in the logic: *Meets*, *Overlaps*, *During*, *Starts*, *Finishes.*

Towards the meaning of these operators, let p and q be predicates. Let p- indicate the unique time when p becomes true and p+ the unique time when p becomes false. Similarly for q. Using this convention, the meaning of the above operators can be defined as follows

- p *Meets* q $\Rightarrow$ p+ = q-
- p *Overlaps* q $\Rightarrow$ (p- < q-) & (p+ > q-) & (p + < q+)
- p *During* q $\Rightarrow$ ((p- > q-) & (p+ <= q+)) or ((p- >= q-) & (p+ < q+))
- p *Starts* q $\Rightarrow$ (p- = q-)

- p *Finishes* q $\Rightarrow$ (p + = q+)

From the above definition it is clear that if p *Finishes* q then q *Finishes* p and if p *Starts* q, q *Starts* p. Figure 2.2 gives a pictorial description of the definition.

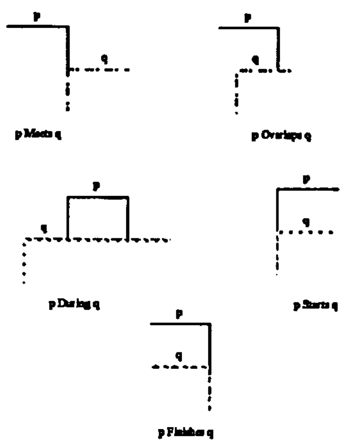


Figure 2.2. Interval Logic

Interval logic, while mathematically sound, has problems when systems based on it are implemented. Only the operators *During* and *Overlaps* are implementable. It is impossible to implement *Meets, Finishes* and *Starts* accurately as it is impossible to verify and guarantee the equality of time as reading the time usually takes non-zero time. Hence, specifications involving operators based on equality will almost always be violated.

The above interpretation of *During* and *Overlaps* is well defined only for a system where all the reasoning is done using a single clock but is not well defined in a system with multiple clocks. For example, a predicate p with respect to clock 1 could be during another predicate q with respect to clock 2 but p with respect to clock 3 may not be during q with respect to clock 4, e.g., because of non-synchronous clocks. Hence, interval logic by itself cannot be used to specify distributed systems unless one extends the semantics of the operators or one works with completely synchronized clocks. However an interval characterization of time is often essential. The semantics of ARL has been designed so as to support both an integer and an interval characterization of time. This will become clear when the semantics of the constructs using time are defined in chapter 5.

The limitations of classical temporal logic have been overcome in logics like RTL [46] etc. We have designed constructs which are similar to suit our language. This is discussed in detail in chapters 4 and 5.

Apart from the theoretical advantages of using a temporal logic, an added advantage is that one need not specify priorities of a scheduling algorithm. The onus of satisfying these temporal constraints lies with the implementation. In other words, the scheduling is abstracted from the program. As we do not wish to curtail the expressiveness of the logic, a large class of temporal requirements can be specified in the language. The only restriction we place on the operators is that they lead to executable specifications.

2.5 Examples of Real-Time Languages

In this section, a few languages designed primarily for real-time systems are discussed. The purpose of this exercise is to identify the advantages and shortcomings of various languages. This enables us to use the results of prior work. We have identified the various drawbacks present in other languages and rectified them when we designed our language.

As it is not feasible to look at all languages designed for real-time systems, some popular classes have been identified and the selected languages are typical of the particular class. The classes chosen are *applicative*, *logic*, *imperative* and *data-flow*. For the sake of clarification, we consider data-flow as a special form of the imperative paradigm.

2.5.1 Applicative Languages

2.5.1.1 ART

Broy in [14] discusses an applicative real-time language called ART. Time in ART is mapped onto the set of natural numbers. Every object is given a time stamp representing the time it was created. The language constructs are function declaration, function application, conditional statement, delay statement (*delay* E1 *for* E2), ordering statement (E1 *before* E2). The delay statement returns the value of E1 after delaying for the time specified by the value of E2. However, the time to evaluate E1 and E2 are also important. Therefore the delay is the maximum of time required to evaluate E1, the time to evaluate E2 and the value returned by E2. The *before* statement forces the evaluation of E1 before E2.

Broy discusses a denotational semantics for the language. To discuss a subset of the semantics, let V_{nrt} be a non real time semantic function typed as: V_{nrt}: expressions (EXPR) $\rightarrow$ environments (ENV) $\rightarrow$ some semantic domain D. Let DOM be D $\times$ Naturals, where the second field of the tuple represents the time stamp of the object in question. Let $\perp$ represent undefined. A real-time semantic function V based on V_{nrt} can be defined as V : EXPR $\rightarrow$ ENV $\rightarrow$ DOM. As an example, the meaning of the delay construct is as follows: Let V(E1)(e) = (o1,t1) and V(E2)(e) = (o2,t2), where o_i is the value of E_i in the semantic domain while t_i is the time stamp of o_i. Then

$$V(\mathit{delay}\ \mathrm{E1}\ \mathit{for}\ \mathrm{E2})(\mathrm{e}) = \begin{cases} (\perp, \infty) & \text{if t2} = \infty \\ (\perp, t2+1) & \text{if o2} = \perp \\ (o1, max(o2, t1, t2)) & \text{otherwise} \end{cases}$$

The above equation indicates that if E2 takes infinite time to evaluate, the meaning of the delay statement is undefined (denoted by $\perp$), i.e. no further information is available about the statement, and takes infinite time to complete. If the evaluation of E2 is undefined, i.e. evaluation of E2 results in an error, the meaning of the delay statement is undefined and this is known just after E2 has been evaluated. In all other cases it is the value of E1, with a time stamp of the maximum of t1, t2 and o2.

Although it is theoretically appealing, ART does not have all the constructs for program development. The language is purely functional and does not consider constructs for nondeterminism, concurrency and distribution. Though one can specify the order of evaluation, using these primitives it is difficult to specify the occurrence of something at an absolute time. Another problem is that delay as defined is realizable only for a single thread of control. If more than one thread is active, the scheduling overhead must be incorporated. Thus, the only realistic interpretation of delay in ART is that of a lower bound. This by itself is not acceptable as one must be able to specify the upper bounds on the delay.

2.5.1.2 Arctic

Arctic [20] is a language designed mainly for real-time control. It is a "stateless" language in which the relationships between system inputs, outputs and intermediate terms are expressed as operations on time varying functions. This characterization, while correct for many of the control oriented systems, is not sufficiently general.

Time is represented by a function called *time*. The resolution of this function is given by *dur*. Basic temporal constraints are specified by the time *at* which a particular "event" is to occur and how long the event is to take to execute. X(*at*)t states that an event X is to occur exactly at time t, while X(*for*)d states that the event X is to take d units of time to complete. It has temporal constructs like *shift* and *stretch* to construct other constraints. The *shift* operator takes an argument and translates a temporal specification by the argument times *dur*. For example, if X is to occur at time t, X *shift* t' forces X to occur at time t + t' × *dur*. The *stretch* operator changes the value of *dur* which was by default 1. For example, X *dur* 0.5 requires X to take half the previously specified time to complete.

For example, chimes causes[E shift 0, C shift 1, D shift 2, G shift 3, G shift 8] specifies the ringing of the bells. When the procedure chimes is called, the E note is played immediately (shift of 0), the C note played 1 unit of time after the call, D after 2 units of time etc.

It is our belief that these operators are not sufficient to model real systems. For example, it is not possible to specify the bounds within which an event is to occur. As mentioned earlier, it is difficult (if not impossible) to implement the *at* or the *for* operators correctly. The requirement that the intermediate terms are also to be expressed as operations on time varying functions is too restrictive. In real-time systems it is not essential to synchronize every single term. It is only certain key terms that have to satisfy timing requirements. The behavior of the program when certain timing specifications are violated is left undefined. If a language has to be used in programming real-time systems, it must facilitate recovery actions when timing conditions go awry.

2.5.2 Logic Languages

The only logic real-time programming language we encountered in the literature was [30]. Systems are described by events and states. States are properties of the objects composing the system and represented by atomic formulae. The rules for describing the behavior of the objects are deductive formulae of the if-then variety whose premises can be events and states and relations between them and whose conclusions are single events or single states. This allows the writing of each rule as a Horn clause.

Events occur at a particular time and time is parameter in the formulae. For example, arrival(L,F,T) represents the arrival of lift L at floor F at time T. As states are properties of objects, they have a duration. All predicates referring to states have two time parameters, which stand for time interval over which the property holds. For example, moving(L,F,D,T1,T2) states the movement of lift L from floor F in direction D holds in the time interval [T1,T2).

Clearly the authors have tried to specify real-time systems within the logic programming framework. Control information, as in Prolog, is specified outside the logic and complicates the understanding of programs. In chapter 6 an example of a program in real-time Prolog is presented. It will become evident that the logic approach requires associating time with most, if not all, objects. This is not acceptable in a language aimed towards prototyping systems.

2.5.3 Imperative Languages

2.5.3.1 Crash

Crash [97] is a real-time language which uses shared variables as the medium of communication and for synchronization. It has *analog* variables to model history of a computation. It has timing construct to express scheduling constraints. Special subprograms called tasks can be defined. Only tasks can be scheduled. The six basic scheduling statements are: *start* a task *immediately*, *start* a task at a particular *absolute time*, *start* a task *within some interval*, *execute* a task *periodically*, *start* execution of a task when a *particular condition is satisfied* and *cancel* an active task.

It is quite well suited for an environment where well designed experiments are to be conducted. It does not support any prototyping features, nor does it consider execution in a distributed environment.

2.5.3.2 Real-Time Euclid

Real-Time Euclid [50] is a language designed specifically to enable one to build reliable hard real time systems. In order to make the system reliable it is necessary to analyze the schedulability of programs and guarantee that in the execution of the program, all deadlines shall be met.

The language is based on the imperative paradigm, and in order to guarantee the schedulability of programs written in it, it eliminates all dynamic features. This includes dynamic arrays, pointers and recursion. As the timing constraints are closely tied with the control structure, temporal analysis of programs is more difficult than were it separate from the control of the program. This coupling also hinders modification to programs as any additional timing condition requires following the control flow of the program. Real-time Euclid can not be considered a language for prototyping real-time systems.

2.5.3.3 Ada

Ada [1] is a language which comes closest to addressing a number of issues in real-time distributed programming. It facilitates modular, parallel programming, data abstraction and is a good tool for implementors. We do not consider it an appropriate specification tool, as there are a number of shortcomings.

The language is not completely defined with respect to distribution. Various attempts to provide a distributed environment have been made [21], [48], [47], [95] and [93]. But there is no agreement regarding the unit of distribution and how distribution of a single program is to be specified. That is, there is no single technique in distributing Ada. This is because Ada does not distinguish between concurrency and distribution. The concept of time which is critical to real-time applications is not complete in Ada as discussed in [96] [94]. Difficulties associated with priority inversion which causes scheduling difficulties has beed identified in [34]. Another drawback is that it does not have a well defined formal model associated with it.

2.5.3.4 ESTEREL

ESTEREL [11, 10] is a synchronous real-time language. It supports programming constructs like modules, types and control structures. The basic communication unit is *signal* which has a name and a value belonging to a particular type. Conceptually, an *event* is an instantaneous broadcast of information. An information can be composed of many simultaneous signals which themselves could convey values.

The *emit* instruction generates a signal and its associated value. The *upto* instruction defines a temporal scope for a code body. The *upto* instruction defines the termination condition for a body in terms of when signals occur. For example, *do* B *upto* S(X) executes B. Whenever the signal S occurs the execution of the *upto* block is terminated and the value associated with the signal is stored in X and the execution continues. As the communication paradigm is synchronous, the occurrence of S, is checked before executing B. If B is a loop, the loop is executed until the signal occurs. The occurrence of an signal causes the loop to exit after the current iteration.

Time is related to the signal flow and there is no universal time reference. It supports classical programming language constructs like modules, types and other control structures. In chapter 6, where we compare ARL and ESTEREL, an example of an ESTEREL program is given.

The language has three types of semantics viz., static, behavioral and computational. The static semantics is used to detect temporal paradoxes, while the behavioral semantics is used to verify program equivalences and the computational semantics is used to specify what the program achieves. The three semantics are discussed in detail in the next chapter.

The main drawback of ESTEREL as in Real-time Euclid is that the temporal specifications are closely tied to the control structure of the language. It complicates the process of making changes to an existing program. It also makes temporal analysis difficult. This will be clear when we discuss examples in chapter 6.

The designers of ESTEREL claim that they wanted the language to be *synchronous* in that a delay statement terminates exactly when its ending event occurs. They also assume that all control transmission like the sequencing operator and simple operations like assignment addition etc., take no time at all. It is clear that these conditions are unrealistic and should not be assumed. The semantics of these operators along with the semantics of signals leads to confusion. For example, as discussed in [10], await S(X); await S(X) does not wait for two signals but only one signal as the ';' operator takes 0 time and the signal is still present when the second await is executed.

Another drawback of synchrony and tying the temporal specifications to the control flow, is that an signal occurrence need not affect the execution of the program 'immediately'. The *upto* statement checks for signal occurrence only when the end of the body is reached.

Note that we are interested in a language for rapid prototyping, which is not a purported goal of ESTEREL. Thus, they do not discuss features like polymorphism, type inferencing, higher order functions, lazy evaluation, declarative constructs etc. They also do not discuss distributed execution of ESTEREL programs.

2.5.4 Data Flow

2.5.4.1 Real-Time Lucid

Real-Time Lucid is a declarative language by Faustini et al [28] based on Lucid [6]. The basic data structures are streams. Streams are constructed via the *fby* operator. For example, x := 1 *fby* (x + 1) defines a stream of positive integers.

Other stream manipulating functions are *asa* and *next*. Asa is a binary filter and takes in two streams, a data stream and a boolean stream. It returns a value corresponding to the index of the first true on its boolean stream. Next is an unary operator and it returns the stream after the first element has been removed.

The language assumes the existence of a global clock. The primitives for real-time programming include specifying the bounds on the time necessary to create and destroy objects. The bounds are specified as time intervals. For example, let x be an object with x_{ct} the permissible creation interval [a,b], and x_{dt} the permissible destroying interval [c,d]. If x_{act} and x_{adt} are the actual creation and destruction respectively, then $a \leq x_{act} \leq b < c \leq x_{adt} \leq d$.

The following is an example program in Real-Time Lucid. It samples an incoming radar signal which can have a value in the range 0 to 100. The goal of the program is to inspect the incoming signal at a specified sampling rate (Sampling) and to output 0 if the signal value is less than 50, 1 otherwise. The output must remain for a fixed duration (Duration) of time.

// Time constraints
stream = < [0,0], [1,1], [2,2] ... >
Radar_{ct} = stream/Sampling
// creation time for each element in Radar must satisfy sampling period
$\text{ZeroOne}_{ct} = \text{Radar}_{ct}$
// Data Dependencies
ZeroOne = Radar >= 50 ;1 ; 0

The only time related constructs are creation and destroying intervals for objects. No other temporal features are discussed. Specification of a recovery action in case of timing error is not addressed in the paper. The authors admit that more work has to be done before it becomes an usable language. The main shortcoming of this approach is that real-time foundation of the language is weak. It assumes that one has knowledge of the life time of all objects. This assumption is not necessarily true in a prototyping environment. If the language has to handle the distributed case, it is necessary to extend the concept of time.

2.5.4.2 Lustre

Lustre, as defined in [9] and [16], is a synchronous data-flow language for real-time systems. The basic data structure is a stream and the operators are very similar to the ones in Real-Time Lucid. A novel idea is the use of boolean variables as clocks. This notion of clocks is not to measure time but rather to be used as a selection function to construct new streams by using the *when* operator. They implicitly assume that an integer timer is present. For example, if a clock C has values true, false, true, true, false, false and true in the first seven instances of time and if the corresponding values of a stream E were el, e2, e3, e4, e5, e6 and e7, the values of the new stream constructed by *when* C(E) is el, el, e3, e4, e4, e4, e7. The value of new stream changes only when the clock is true.

Functions *count* and *rank* using this notion of clocks are defined. If C a clock and n an integer, Count(C,n) is equal to the number of units of time the clock was true, while Rank(C,n) is equal to the number of units of time say m, to be considered to make Count(C,m) = n. These functions help the definition of a denotational semantics for Lustre.

The computation of a rising edge of a variable is shown below. The function EDGE takes in a boolean stream as input an returns a boolean stream. A rising edge is detected when the current value (C) is true and the previous value (pre(C) is false. The Lustre program is

```
EDGE(C: boolean) return (H: boolean) ;
H = C  - (C and not pre(C))
```

Clocks and other related statements in Lustre are more like ZF iterators of Miranda [88] as they construct new sequences from old. Boolean sequences are considered to be clocks and are constructed by the program. It is our feeling that the notion of clock should be independent on the flow of the program. In a real-time language, clock(s) should influence the flow of control and not the other way.

Like ART and Real-time Lucid, Lustre has very few programming constructs, which the authors admit makes it difficult to specify realistic systems.

Conclusion

In summary, our brief survey has shown that if a language has a formal basis it is too restricted to be used a rapid prototyping language, while if the language has a large number of programming constructs there is no discussion of an associated formal structure. None of the languages permitted the specification of recovery in the case of timing violations. Also, none of these languages were designed to prototype real-time systems.

One can conclude that a programming language which addresses the issues in building distributed real-time systems along with an associated model is essential. In the next chapter we discuss the various formal models developed to characterize concurrency and real-time.

CHAPTER 3

SURVEY OF SEMANTIC STYLES FOR CONCURRENT/REAL-TIME SYSTEMS

There have been a number of approaches to provide semantics for programming languages. Two of the most prevalent are the denotational and the operational styles. A brief introduction to denotational semantics is presented. Its limitations and our preference for operational semantics is discussed. Following this, various approaches to developing operational semantics are discussed.

3.1 Denotational Semantics

Denotational semantics gets its name from assigning *denotations* — which are abstract entities modeling meaning — to language constructs. We describe this briefly. The following is by no means a comprehensive and a fully technical discussion. The interested reader can find the details in [35]. For the sake of simplicity we restrict our attention to *sequential languages.* [13, 55] describe denotational semantics of concurrent/real-time systems.

Denotational semantics is composed of functions associating elements in the syntactic domain (like assignment, expressions subprogram calls) to elements in a well structured domain. The first step, in denotational semantics for an imperative language, is usually to formalize the concept of state. For example, states could be represented by a set of identifier value pairs. A state could also be a set of identifier value pairs along with two sequences of values representing input and output performed by the system. Note that this is sufficient as we are considering only a sequential language. Towards formalizing this we define the syntactic domain first. Let **Ids** be the set of identifiers, **E** the set of expressions and **Cmd** the set of commands allowed by a language. For semantic considerations assume that there is a set of expressible values called $\mathcal{EXP}$. Also assume that the set $\mathcal{ENV}$ represents the environment, which can be defined as the function space from **Ids** to $\mathcal{EXP}$. In other words, environment as used in denotational semantics, defines values for identifiers. To define the meaning of elements in **E** we construct a mapping Ξ: **E** $\rightarrow \mathcal{ENV} \rightarrow \mathcal{EXP}$. In other words, the meaning of a given expression is a function from an environment to the set of expressible values. The environment gives values to the identifiers in the expression, while the element in the set of all possible values represents the value of the expression. Similarly towards the meaning for **Cmd** assume a set $\mathcal{S}$ to be set of states. Define a meaning function Θ: **Cmd** $\rightarrow \mathcal{S} \rightarrow \mathcal{S}$. Intuitively, commands transform a state to another state.

Mathematically speaking, denotational semantics is a homomorphism from the algebra of syntax into an algebra of semantics. The syntactic algebra consists of the permissible syntactic constructs while the semantic algebra consists of semantic domains and interpretation of the operators in the algebra. The semantic functions together form the homomorphism.

We do not discuss denotational semantics in any more detail as we are not currently interested in it due to certain limitations. These shortcomings are discussed below.

3.1.1 Limitations of Denotational Semantics

The limitations of denotational semantics as developed in [35, 84] are discussed in [71, 66, 98]. These limitations can be summarized as follows. Denotational semantics is based on what are called λ expressions. The semantics of trying to model environments, stores etc. in lambda calculus gets intertwined with the semantics of the programming language it is trying to explain. The notion of environments etc., should have a natural representation in the semantic technique used if the semantics is to be simple. Understanding λ programming, especially when it involves non-obvious constructs (like continuations), requires a lot of effort. In order to minimize the information necessary to specify the transformation, many constructs like routine entry, exit, the distinction between parameters and variables etc., are expressed in not too understandable terms. While these principles are mathematically sound, their relevance to the ordinary computer scientists is not apparent. To quote Dana Scott [78]

> ... The difficulty in the presentation of the subject is in justifying the level of abstraction used in comparison with the payoff: too often the effort needed for understanding the abstractions does not seem worth the trouble — especially if the notions are unfamiliar or excessively general.

This is especially true if the primary concern of the reader is in implementing the language. As we have designed a language to support rapid prototyping comprehensibility and implementation of the language have a high priority. Thus, denotational semantics are not quite appropriate at this time.

Operational semantics (which is described below) is better suited for the purposed of deriving a semantics based implementation. This however is not to say that denotational (or any other abstract) semantics is not relevant. It is only because the focus of this thesis is in designing a language and describing it formally so as to aid an implementor, that we do not discuss denotational semantics. Other activities like development of a proof theory for the programming language, verification techniques for the language etc., a denotational definition is desirable.

3.2 Operational Semantics

Operational semantics can be viewed as associating an abstract machine with the language in question. The behavior of the machine describes the effect of executing a program. This definition of operational semantics permits an implementation (compiler/interpreter) of the language to be an acceptable definition of the operational meaning of the language. Needless to say, this definition is at a very low level. One of the first high level operational semantics styles was the use of Vienna Definition Language(VDL) [99].

3.2.1 Vienna Definition Language

The Vienna Definition Language (VDL) [99] defines an ideal machine which executes the program. It is a tree based technique. That is, the transition rules transform one tree to another. An algorithm to convert a program into its abstract form similar to a parse tree, is assumed. The abstract program with the initial data determines an initial state. This initial state is transformed into other states by the transition functions, until a final state or a state which has no successor is reached.

Formally, a Vienna definition system V is defined as V=(EO,CO,Ω,S,σ,μ) where EO is a set of elementary objects, CO is a set of composite objects and is disjoint from EO, $\Omega \in$ EO is a distinguished element called the null element, S $\subset$ EO is a set of elements called simple selectors, σ and μ are operators called the selection and assignment operators. σ takes a selector and an object and returns an object (which could be null). μ takes an object, a selector and another object and updates the field indicated by the selector of first object to the second object. For example, a tree with three selectors (or branches name) s1, s2 and s3 with sub-trees x1, x2 and x3 respectively can be represented as μ_0(<s1:x1>,<s2:x2>,<s3:x3>), where μ_0

represents the tree forming operator. Call this object t. σ(t,s1) is x1. Altering the second sub-tree to y is achieved by the command μ(t,s2,y).

Every computation starts from an initial state/representation, and is characterized by a sequence of states or information configurations. A state is represented as a composite object whose components may be selected by the selectors. An information configuration is obtained by applying an instruction (transition rule) to the previous configuration. These actions are constructed using the assignment operator. An instruction of a VDL machine has the form

```
name(p1,p2,p3 ... pn) = [
        $c_1 \rightarrow a_1$
        $c_2 \rightarrow a_2$
        $c_3 \rightarrow a_3$
          ⋮
        $c_m \rightarrow a_m$ ]
```

The p_i's are parameters, c_j's are boolean valued conditions and a_j's are the actions to be executed if the boolean valued condition corresponding to it evaluates to true. The condition action pairs are treated as a sequence and the action corresponding to the first true condition is executed.

Consider the following example which is an outline of the transition rules to evaluate a term. Call the evaluation routine *value*. Also assume the definition of a function called *apply* which applies an operator to its arguments and the object *env* representing environment. Functions *is_binary*, *is_var* and *is_const* which identify a term as an application of a binary operator (defined by the op selector), a variable and a constant respectively are assumed. Towards the transition rule for *value* define, a to be *value*(σ(s1,term)) and b to be *value*(σ(s2,term)), where s1 and s2 are selectors. The transition rule is

```
value(term) = [
        is_binary(σ(op,term)) → apply(a,b,σ(op,term))
        is_var(term) → σ(term, env)
        is_const(term) → term]
```

The above rule states that if the term 'term' is formed using a binary operator 'op', the value of the term is the application of it to a and b. If 'term' is a variable, the value is given by its binding (*env*) and if 'term' is a constant the value is 'term' itself.

VDL supports the concept of state consisting of store and control. The complexity of the store depends on the language for which the model is being used. For example, it can be a set of name value pairs, or could consist of environments to cater to block structured languages. The control aspect contains an abstract program along with instructions for the interpreter.

3.2.2 Structured Operational Semantics

The main limitation of operational semantics based on VDL was that it was overly verbose. These techniques were not considered to be "directly formalizing the intuitive operational meaning found in most language definitions." For instance using the VDL machine to describe a language results in a large number of transition rules that deal with the manipulation of the tree structure. In other words, the VDL machine has a preconceived notion of what data structures and operations should be used to implement the language. Similarly, the transition rules of the SECD machine most of the instructions were to manipulate the stack.

Plotkin in [72], proposed a higher level of operational semantics, which employed axiomatic descriptions of the transition rules. In this approach a number of transitions were coalesced to form a single transition rule with more meaning. The intermediate steps of manipulating the various data structures is ignored. Apart from reducing the number of transition rules, it also permits the implementor to choose the appropriate data-structures. The only restriction is that the behavior exhibited by the chosen data structures must satisfy the defined axioms.

Define a configuration as a two tuple with the first field an expression or a statement and the second field the state of the memory in which the statement/expression is to be executed/evaluated. The transition rules, as in VDL, form a relation on a set of configurations. The transition rules consist of two parts: the antecedent and the consequent. The antecedent and consequent can be considered as the conjunction of subparts separated by commas. While the antecedent and the consequent have the usual logical interpretation, they can also be interpreted as rewrite rules. The following examples will illustrate the technique. The first rule captures the intuitive meaning of statement composition.

$$\frac{<\text{c1}, \text{s1}> \longrightarrow \text{s2},\ <\text{c2}, \text{s2}> \longrightarrow \text{s3}}{<\text{c1;c2 s1}> \longrightarrow \text{s3}}$$

The rule is to be interpreted as "if the execution of command c1 in state s1 alters the state to s2 and the execution of command c2 in state s2 alters the state to s3, the execution of the composition of c1 and c2 in state s1 results in state s3." The two rules described below define the meaning for the 'while' statement. Notice that the rule described above had two components to the antecedent while both the rules described below have only one component.

$$\frac{\texttt{<b,M>} \rightarrow \texttt{<true,M'>}}{\texttt{<while b do c, M>} \rightarrow \texttt{<c;while b do c, M'>}}$$

$$\frac{\texttt{<b,M>} \rightarrow \texttt{<false,M'>}}{\texttt{<while b do c, M>} \rightarrow \texttt{<null, M'>}}$$

The Plotkin style operational semantics consists of giving a set of rewrite rules which transform a program step by step into its result. This technique has been then applied to CSP and is described in [73].

The two techniques described above (VDM and SOS) are very general and not domain specific. They can be applied to any type of system including distributed real-time systems. However, the characterization of parallelism in such models is far from obvious as they do not have operators for parallelism (or concurrency.) A few operational styles designed specifically for concurrent computation are studied below. These techniques can be thought of defining formal languages which are programming language independent. It is possible to use these formal languages as a target language for a concurrent programming language. The translation will provide the operational semantics for the programming language.

3.2.3 Calculus for Communication Systems

Calculus for Communicating systems(CCS) [64] is a paradigm for concurrent computation. Concurrent computation is modeled by communication (or events) between independent processes. It assumes that a system is observable in that the events the various processes generate can be noted. The system can also be thought of as one on which certain experiments (or actions) can be performed (by generating an event from outside) and studying the system's response. If the system rejects an experiment it gets 'stuck' i.e., no further progress is possible. In other words, a rejected experiment causes the system to come to an abnormal halt. The system is described by the set of sequence of experiments it admits without getting stuck.

A system description consists of a set of triples called a labeled transition system. The meaning of a typical triple (s,a,s') is that the process s is transformed to process s' by performing action 'a' on the environment or by accepting an 'a' experiment. We write s $\xrightarrow{a}$ s' to represent (s,a,s'). Non-determinism is expressed when more than one triple is active. For example, let s be a process on which actions 'a' and 'b' can be performed (i.e., (s,a,s') and (s,b,s") can be executed.) Assume that both events a and b are available. The existence of the two rules can be interpreted to mean that the system being described can non-deterministically choose one of the two transitions. The labeled transition technique describes a process by generating a labeled graph. See figure 3.1. A collection of such graphs describes a concurrent system.

CCS defines a set of operators which are used to construct processes. The set of actions that can be executed by the processes forms the basis of a state. Towards a syntactic definition of the operators in CCS, let Δ be a set of *actions* (or experiments), $\mathcal{V}$ the set of processes. Assume that there is a bijection on the elements in Δ. Let μ be a typical element of the set of *actions* with $\overline{\mu}$ be the corresponding element under the bijection called complementary action. Complementary actions are used to denote actions which occur in pair like sending and receiving etc. Let E be an element of $\mathcal{V}$, i.e., represent a process. Let a *null* process be a distinguished element of E. The following is a syntactic description of the operators. The set of processes are defined recursively, i.e., they describe derivation of new processes from old.

- $\mu \cdot$ E denotes the acceptance of action μ after which the process behaves as defined by E.
- $E_0 \mid E_1$ is called composition which is defined below.
- $E \backslash A$ $A \subseteq \Delta$ denotes restriction of actions to elements *not* in A.
- E[S] , where $S : \Delta \rightarrow \Delta$, a relabeling of actions.
- Σ E_i, called summation which is also defined below.

Informally, composition allows concurrent behavior of E_0 and E_1 with communication through complementary actions, while restriction using set A, eliminates the experiments involving elements belonging to A, i.e., the derived process does not accept any event specified in the set A. Summation denotes a non-deterministic selection of one of the components. Summation of two terms (binary summation) is denoted by '+'. The above syntactic definitions satisfy the properties described below. Note that certain properties are written in the form of a rewrite rule, viz. a horizontal line separating the antecedent and the consequent.

- $\mu \cdot E \xrightarrow{\mu} E$
- $\dfrac{E_i \xrightarrow{\mu} E'}{\Sigma E_i \xrightarrow{\mu} E'}$
- $\dfrac{E_0 \xrightarrow{\mu} E_0'}{E_0 \mid E_1 \xrightarrow{\mu} E_0' \mid E_1}$

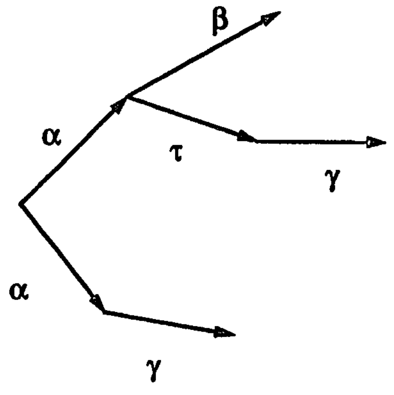


Figure 3.1. Tree Representation of CCS program

- $\dfrac{E \xrightarrow{\mu} E'}{E \backslash A \xrightarrow{\mu} E'}$ iff $\mu \notin A$.
- $\dfrac{E \xrightarrow{\mu} E'}{E[S] \xrightarrow{S(\mu)} E'[S]}$

From these rules of action, the behavior of a process can be represented as a tree. For example a process specified as $\alpha \cdot (\beta + \tau \cdot \gamma) + \alpha \cdot \gamma$ can be represented by the tree shown in figure 3.1. It defines a process which after action α has a choice of either behaving like $(\beta + \tau \cdot \gamma)$ or like γ. If the former is chosen, the process has a choice of either performing action β or $\tau \cdot \gamma$.

Milne in [63] has shown how it is possible to represent the ticking of time (global clock) in an extended CCS framework called Circal. As the increment of global time should not depend on the behavior of the program, the definition of each of the processes is augmented with a choice sum involving an action τ which signifies the ticking of time. A single tick occurs if no action takes place. If an event occurs, it does so simultaneously with a clock tick. Simultaneous occurrence is indicated using '(' ')'. For example, $(\tau \cdot e)$ indicates the simultaneous occurrence of the ticking of time and occurrence of action e.

In the example, there are three processes A, B and C. Consider γ as an event representing the transfer of information from C to A, α as an event representing transfer of information from A to B and β as an event representing transfer of information B to C. When a process receives information, it sends it out (on its output channel) at the next instant. If no input is received, time ticks. This continues for ever and hence the processes are defined recursively with <= denoting definition.

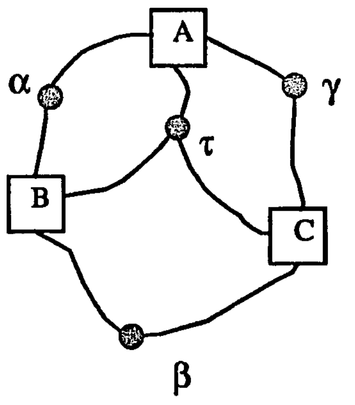


Figure 3.2. Time in Circal

See figure 3.2 for an interconnection between A, B, C and the clock. The Circal notation for the above description is as follows.

$$\begin{array}{l} A <= (\gamma \cdot \tau)\cdot(\alpha \cdot \tau)A + \tau\cdot A \\ B <= (\alpha \cdot \tau)\cdot(\beta \cdot \tau)B + \tau\cdot B \\ C <= (\beta \cdot \tau)\cdot(\gamma \cdot \tau)C + \tau\cdot C \end{array}$$

As CCS (or CIRCAL) has a formal semantics, a language designer can translate the constructs of the language in question to CCS. The operational nature of CCS induces an operational semantics for the language. This technique has be applied to a subset of Ada i.e., a subset of Ada has been translated into CCS [41].

3.2.4 CSP

The CSP [44] model is similar to the CCS model, in that there are events/actions and channels over which communication occurs. Unlike CCS where the emphasis was on communication, the main feature of CSP is a process. A process is defined to be a behavior pattern of an object describable in finite terms. Events or actions are used to describe the behavior of processes. The main operators are:

- $\rightarrow$: Next
- μ: Parametrization
- $||$: Parallelism
- "?": Channel Input

- "!" : Channel Output
- $\sqcap$: Non-determinism
- $\square$: Controllable choice

The principal operator in the description of a process is the $\rightarrow$ operator. It plays the role of $\cdot$ in CCS. Let 'a' be an event and P be a process. a $\rightarrow$ P describes a process which engages in event a and then behaves like P. Process can also be defined recursively. For example, P = a $\rightarrow$ P describes a process which engages in an infinite sequence of a's. The reader may assume the a's to be ticks of a clock, to get a feel for situations where such descriptions might be useful. Processes can be parameterized using the μ operator. For instance, P(x) = μx: x $\rightarrow$ Q can be instantiated with any event (say 'a') which is substituted for x yielding P(a) = a $\rightarrow$ Q. Processes which are to be executed in parallel are composed using the $\|$. P $\|$ Q describes a process composed of two sub-process P and Q where P and Q can execute in parallel. Communication is a special form of an event. The event is a two tuple involving a value and a channel. A process can wait for a value to occur on a channel by using the "?" operator. For example, c?x describes a process that is waiting for a value to be sent on channel c and when the value arrives it is stored in the variable x. Values are transmitted on the channel using the "!" operator. c!v emits the value v on the channel c. As CSP supports a synchronous communication paradigm, the statement does not terminate unless there is a reading of channel c using a c?x.

CSP (like CCS) supports two types of non-deterministic operators. One is called non-determinism and the other controllable choice. Non-determinism is represented by $\sqcap$ and controllable choice is represented by $\square$. In non-determinism the environment has no knowledge and no control over the selection of the process to be executed. In controllable choice, the environment can control which of the processes involved is selected, provided the control is exercised on the *very first action*. The following law formalizes the above description.

$$(x : A \rightarrow P(x))\square(y : B \rightarrow Q(y)) = \begin{cases} z : (A \cup B) \rightarrow & \text{if } z \in (A - B) \text{ then } P(z) \\ & \text{elsif } z \in (B - A) \text{ then } Q(z) \\ & \text{elsif } z \in (A \cap B) \text{ then}(P(z) \sqcap Q(z)) \end{cases}$$

In the above equation, the notation of x : A$\rightarrow$ P(x) is to be read as x from A then P of x. In other words, it defines a process which first offers a choice of any event x in the set A and then behaves like P(x). The law described above states that if an action solely in A can be performed then it is, and similarly for B. If an action which is common to both A and B then the selection of which process to execute is not specified by the axiom i.e., is non-deterministic.

Other operators to specify interleaving, checkpointing, interrupts are present. The development of these operators requires the introduction of a number of definitions. We refer the reader to [44]. [44] also discusses the properties and laws of all the constructs in detail. As in CCS, the existence of an operational semantics for CSP implies that a translation of a language into CSP defines an operational semantics for the language.

Reed and Roscoe in [75, 76] discuss a timed extension of CSP. They use a dense notion of time and extend CSP with a WAIT construct. WAIT t terminates after t units of time. The meaning of a CSP process has to be augmented to know when a process will be ready to handle actions or respond to the environment. This is called the *stability* as it represents the stage after which the process cannot make any more internal progress. In other words time stability represents the time after which a process which has the exhibited current trace s will be ready to respond to the environment, i.e. can handle an event. Towards the definition of stability let time be represented by T and Σ be the set of events/actions. The set of ordered timed traces is defined as $\{s \in (T \times \Sigma)^*$ such that if $(t\ a) < (t'\ a')$ then $t < t'$ $\}$. It represents the occurrence of events ordered by time. Also define the *refusal* of a process as the set of events which the process can fail to engage in over a specified time interval. In other words, it represents the set of events that the process will not accept over the specified interval. Model a timed CSP process as a set of ordered three tuples $(s,\alpha,\aleph)$, where s is a timed trace of the process, $\aleph$ is a *refusal* of the process and α is the time at which the process is guaranteed to be *stable* after exhibiting the trace s and refusing $\aleph$. This forms the basis of a configuration on which the semantics of timed CSP is developed. Due to the complexity of the rules, we do not present them here, but refer the reader to [75, 76].

The purpose of the review is to study the effect of adding a time to an un-timed model. It is clear from the brief overview that addition of time complicates the semantics. Essentially, all *relevant* actions have a time associated with them.

3.2.5 S K Reduction Machine

This section is a slight departure from the discussion of general operational models. We discuss an implementation technique for *functional* languages as developed by Turner in [91]. The technique is discussed in great detail in [24]. Our interest in this implementation technique is because it can be thought of as defining an operational semantics. By imposing a left to right interpretation, these equations, could be cast into a rewrite system providing an operational meaning to lazy functional programs. As our language supports lazy computation, it is essential to understand its operational semantics.

The main feature of the technique is the use of combinators or operators that serve the role of an environment, that is they define the binding of variables to values. The compiler transforms each expression into its combinator form which is stored internally as a tree. At run-time, the system executes this combinator code based on a graph reduction algorithm [91]. As it is a *normal order reduction*, the machine automatically supports lazy evaluation, which is necessary to implement ARL.

Three basic combinators **S**, **K** and **I** are defined. They take functions as arguments and return functions as results whose behavior satisfy the following equations.

- **S** f g x = f x (g x)
- **K** x y = x
- **I** x = x

The above equations are to be interpreted as describing the behavior of functions under parameter x. For example, when **S** is applied to functions f and g, a new function say h is obtained. The behavior of h when applied to x, is the same as applying f to x and (g x).

Using these combinators one can verify that the successor function can be defined to be **S(S(Kplus)(K1))I**. The generation of this long winded code is explained in [91] . When this is applied to say 10, the combinator to be reduced is **S(S(Kplus)(K1))I** 10 which can be rewritten to 'plus 1 10' using the above rules. To get more compact code, combinators **B** and **C** are defined. They satisfy the following equations

- **B** f g x = f (g x)
- **C** f g x = f x g

The **B** and the **C** combinators can be re-written in term of **S**,**K** and **I** as follows.

- **S**(**K** E1) E2 = **B** E1 E2
- **S** E1 (**K** E2) = **C** E1 E2

Other combinators are **P** for pairing, **U** to handle functions with more than one parameter (or uncurry), **Y** to handle recursion, **cond** for conditionals. These combinators satisfy the following properties.

- **U** f **P** x y = f x y
- **Y** h = h (**Y** h)
- **cond** true x y = x
- **cond** false x y = y

3.3 Stepwise Development of Operational Semantics

Having reviewed several techniques, the quintessence of each computational feature is examined. By starting with a simple language and adding features like communication, non-determinacy and assignment, the increase in semantic complexity will be observed. This will help the reader in understanding some of the issues discussed in chapter 5. The first language (L_0) has only the parallel construct and local non-determinism. The second language (L_1) introduces channel communication, while global non-determinism is introduced in the language L_2. The language L_3 models real languages with assignment and CSP like channels.

In [22] the denotational and operational semantics for imperative languages is developed in an incremental fashion. Initially, a language with only parallelism but no communication or synchronization is considered. To this language, communications and synchronization are added and the semantics re-developed. This identifies the necessary features in a model to support various constructs. We have culled only the operational semantics part as it is the focus of this thesis. The operational semantics is based on transition systems a la Plotkin [73] with evolution represented by '$\rightarrow$' and the rules written in the rewrite style with a horizontal line separating the antecedent and the consequent. The interleaving model of parallelism is used [22]. Interleaving can be seen as scheduling parallel components on a single processor. The interleaving model also assumes that a basic statement (action) is the unit of schedulability.

3.3.1 Language L_0

The structure of L_0, a simple language without channel communication or global non-determinism, is

$$s ::= a \mid s1;s2 \mid s1 \cup s2 \mid s1 \parallel s2.$$

In other words, a statement s can be recursively defined as one of

- An elementary action denoted by "a"
- Sequential composition (;) of statements
- Local non-determinism ($\cup$) (as opposed to controllable choice) of statements
- Concurrent execution ($\|$) of statements

Towards an operational semantics, let A represent the set of all elementary actions. Let A^* represent the set of all finite strings over A and A^ω represent the set of all strings of countably infinite length over A. Let $\perp$ represent an error (e.g., incomplete information or non-termination with no further elementary action). The set of streams over A, A^{st} is defined to be $A^* \cup A^\omega \cup A^* \cdot \{ \perp \}$. A^{st} is set of all finite words, infinite words and finite words followed by $\perp$. The set of streams forms the basis of the configuration for the operational semantics. Let ϵ represent the empty stream. Let $\cdot$ indicates concatenation in the semantic domain. Let $\perp \cdot a$ be the same as $\perp$.

A configuration for the language L_0 is a pair <s,w> where s is a statement and w an element of A^{st}. 'w' denotes all the actions performed until the execution of the statement s. Assume, for all $w \in A^\omega \cup A^* \cdot \{ \perp \}$ that $<s,w> \rightarrow w$ holds. This axiom states that an infinite or a terminated computation cannot be extended.

Towards a semantics for L_0, define the following axioms for all $w \in A^*$.

$$<a,w> \rightarrow w{\cdot}a$$

$$<s1 \cup s2,w> \rightarrow <s1,w> \mid <s2,w>$$

$$\frac{<s1,w> \rightarrow <s',w'>}{\begin{array}{c} <s1;s2,w> \rightarrow <s';s2,w'> \\ <s1\|s2, w> \rightarrow <s'\|s2,w'> \\ <s2\|s1, w> \rightarrow <s2\|s',w'> \end{array}}$$

The first rule states that an elementary action extends the current stream. The second rule states that any statement in a non-deterministic statment can be chosen. Hence the possible observations is the union of either choice. The two rules defining the parallel operator state that the scheduler decides which process to execute. Using the above rules as inference rules, one can prove that $<(a1;a2)\|b,w> \rightarrow <a2\|b,w.a1>$ by using the first $\|$ axiom and the concatenation axiom.

3.3.2 The language L_1

The language L_1 introduces synchronous communications. Let C be the set of all communications and denote a typical element by c. Augment the alphabet A from L_0 to include C. For the purposes of synchronous communications, a bijection on C is defined such that for every $c \in C$ there is a $\bar{c} \in C$. Define $\bar{\bar{c}}$ to be the same as c. This bijection is to be interpreted as: c can communicate only with $\bar{c}$. Synchronization (i.e., execution of c and $\bar{c}$ simultaneously whose precise definition is captured by the semantics) is represented by τ (in the semantic domain) and is considered to be an element of A. The introduction of communications requires a more sophisticated interpretation of the parallel construct. It is also possible that a statement may fail due to improper communication. Introduce a new symbol δ different from $\perp$ and τ, to indicate failure due to communication. The set of streams (over the new alphabet) is extended to $A^{st} \cup A^* \cdot \delta$, where A^{st} is defined as before. The new term defines communication failure. Towards the semantics of L_1, define the following as before: For all $w \in A^\omega \cup A^* \cdot \{ \delta, \perp \}$, $<s,w> \rightarrow w$, which states that an infinite or a terminated computation cannot be further extended. The new rules are described below.

The first rule is: for $w \in A^*$ $<c,w> \rightarrow w \cdot \delta$ The above rule states that an individual communication fails. That is, for a communication to succeed there has to be a matching communication in a concurrent process. Synchronization in a context is represented by $<c;s1||\bar{c};s2,w> \rightarrow <s1||s2, w \cdot \tau>$. The τ represents successful synchronization. Note that we do not characterize the values exchanged in the communication. Synchronization failure is similar to individual communication and is represented by the following rule $<c||d,w> \rightarrow w \cdot \delta$ if $d \neq \bar{c}$

3.3.3 The language L_2

In the language L_2, global non-determinacy is introduced and denoted by "+". The main feature of global non-determinacy is that the moment of choice is preserved. For instance, s1;(s2+s3) is not the same as (s1;s2) + (s1;s3). The transition rules new to L_2 are as follows.

$<a+c,w> \rightarrow w \cdot a$ and $<(a+c)||\bar{c},w> \rightarrow w \cdot \tau$

The rules state that if there is a choice between an action and communication, the action is taken unless a matching communication is available. However if a matching communication is available, the action *must not* be taken and the communication action *must* be taken. This prevents a process from communication failure. This was not required in the case of local non-determinism where a the choice made in one process is independent of the other processes.

3.3.4 The language L_3

The final language we discuss is L_3 which has assignment, and communication. Communication involves exchange of values as in CSP [44]. To present the syntax of L_3, new syntactic categories namely, variables(Var) and expressions (including boolean expressions) are assumed. The syntax for L_3 has statements s which have one of the following form

- Assignment of the form v := e
- Boolean expression (b)
- Read from channel c?v
- Write to channel c!e
- Sequencing s1;s2
- Global Nondeterminism (or controllable choice) (+)
- Parallel (||)

The reason for having a boolean expression as a statement is that it allows the characterization of certain statements like if, while etc. For example, if b then s1 else s2 is equivalent to (b;s1)+(¬b;s2)

The fundamental feature of the semantics of L_3 is the notion of *state*. The set of states is the function space Σ and is defined to be Var $\rightarrow$ V, where Var denotes the set of variables and V the set of values. We use σ, σ' etc. to denote elements of Σ. Notice that Σ plays the role of A in the previous languages.

For the operational semantics, define the set of streams as $\Sigma^* \cup \Sigma^\omega \cup \Sigma^* \cdot \{\delta, \perp\}$. For an expression e, let $[\![e]\!]$ (σ) represent the value of e in state σ. Let $\sigma(\mathrm{v1}(\sigma)/\mathrm{v})$ stand for a new function derived from σ such that it is different from σ only for the argument v (a variable) in which case it returns v1. Most of the transition rules will be the same as before and are not reproduced. The transition rules which are relevant deal with assignment and communication. The two rules described below state that direct assignment or assignment from channel alters the state as to reflect the new value attained by the variable involved in the assignment.

$$<v{:=}e,\sigma> \rightarrow \sigma(\llbracket e \rrbracket(\sigma)/v) \qquad <c?v||c!e,\sigma> \rightarrow \sigma(\llbracket e \rrbracket(\sigma)/v)$$

Discussion

To summarize, the structure of the semantics has to be enhanced with the addition of syntactic terms. However, the notion of streams is sufficient to characterize meaning given interleaving semantics. Note that the meaning of a program in all the cases will be the set of all possible individual meanings. In other words, the meaning of a program will be set of all possible streams over the appropriate alphabet.

Having discussed some general styles, we review some of the application of these techniques to develop semantics for programming languages is reproduced. We discuss the semantics of an Occam-like language designed for distributed computation, POOL a parallel object oriented language and Esterel, a real-time language.

3.4 Real-Time Semantics for an Occam like Language

Huizing et. al. [45] present a fully abstract (or syntax directed) semantics for real-time distributed systems that is an extension of [55] which they claim was not fully abstract. The principal concern of fully abstract semantics is that the meaning of a construct should be composed of the meaning of the sub-terms of the construct.

The semantics of a program is the set of all histories (i.e., a sequence of information such as messages sent and received, processes waiting to synchronize with others etc.) that can be elicited by the environment in which the program executes. To simplify the semantic model, they assume that time is discrete, every elementary action (assignment, communication, passing a guard) takes a unit time and in a parallel statement all processes start immediately. They also assume the maximal parallelism model where the view is that no unnecessary delays are incurred at any time. This assumption implies that no two processes wait for each other as each process has its own processor and communication is treated as an elementary action (i.e., takes unit time.)

Operational semantics is based on *labeled transition* relations that transform configurations consisting of pairs of statements and states. A state is a function which yields values for the variables defined in the system. The meaning of executing a statement P in state σ is defined by a sequence of transformations into a null statment with the final state being σ'. Note that such a definition is meaningful as P can be compound statement.

All transitions are labeled by L which consists of the set of communications that take place during the step denoted by L^c and the number of local actions (can be considered to be time steps) that are performed during that step denoted by L^n, i.e., L = (L^c, L^n). For example, consider the assignment statement x:= e, where x is a variable and e an expression which is assignable to x. Let σ be a state function. The meaning of the assignment statement executed in state σ can be expressed as (x:=e,σ) $\rightarrow$ ($\emptyset$, (σ(e)/x)) with label ($\emptyset$,1). The label indicates that the assignment was a local (an elementary) action (taking unit time) and involved no communication (indicated by $\emptyset$). Notice that (σ(e)/x) is a new function with one argument such that if the argument is x the value returned is e otherwise it is the value σ(argument).

A more interesting example deals with the parallel construct. Assume that processes P1 and P2 can be composed over (or share) a set of channels (say A.) Denote this composition by P1 $\|_A$ P2. Also assume that processes which can be composed do not share any variables. Denote the restriction of channel labels by $L_1{}^C\backslash A$ and $L_2{}^C\backslash A$. The parallel axiom can now be stated as follows.

$$\frac{L_1{}^C\backslash A = L_2{}^C\backslash A,\ (P1,\sigma)\xrightarrow{L_1}(P1',\sigma_1'),\ (P2,\sigma)\xrightarrow{L_2}(P2',\sigma_2')}{(P1 \|_A P2,\ \sigma) \xrightarrow{L} (P1' \|_A P2',\ \sigma')}$$

where L = (max($L_1{}^n$, $L_2{}^n$), $L_1{}^C \cup L_2{}^C$) and σ' the new state is defined as

$$\sigma'(x) = \begin{cases} \sigma_1'(x) & \text{if x element of var(P1)} \\ \sigma_2'(x) & \text{if x element of var(P2)} \\ \sigma(x) & \text{otherwise} \end{cases}$$

The first condition of the antecedent, requires the communications involved in the parallel construct (i.e. over the set A) to be synchronized. That is, the same set of communications must occur during the parallel execution. The rest of the antecedent is to be interpreted as if P1 in state σ is transformed to P1′ in state σ_1' and P2 in state σ is transformed to P2′ in state σ_2', the parallel composition of P1 and P2 will yield a state σ' as defined. The consequent states that the execution of composition of P1 and P2 over A can be rewritten to the execution of composition of P1′ and P2′ over A in a new state. The transition is labeled L. The time component of L indicates that the time to complete the rewrite of the parallel composition is the maximum of the time taken by the individual processes, while the channel component of L denotes the union of the individual channels.

The main concern in [45] is the fully abstractness of the semantics. From an operational view point, the presentation is no different from the Plotkin style. The main drawback of this technique is that simplifying assumptions regarding the time it takes to statements involving communications are made.

3.5 Operational Semantics of POOL

POOL [5] is a parallel object oriented language. For an example of a typical objected oriented language see [33]. POOL's operational semantics is also based on a transition system. The semantics of a program is presented in terms of the transitions it can make from one configuration to another. A configuration consists of the statement(s) to be executed and the state in which it is to be executed in. Towards a definition of configuration, define *LStat*, for labeled statements, as a set of 2-tuples $<\alpha$,s$>$ denoting that object α can execute statement s, Σ the set of states, *Type* a function which assigns a class (similar to type) to an object and U the set of definitions of classes (similar to abstract types) and methods (analogous to subprograms). Let $\mathcal{P}_{\text{fin}}$(X) denote the set of finite sub-sets of the set X. Define *Conf*, the set of configurations, to be $\mathcal{P}_{\text{fin}}(\textit{LStat}) \times \Sigma \times \textit{Type} \times$ U. Let σ be a typical element of Σ, τ a typical element of *Type* and u a typical element of U.

The transition rules form a relation on *Conf* i.e., are a subset of *Conf* $\times$ *Conf*. Based on the definition of *Conf*, the axioms satisfied by the various syntactic constructs can be specified. The axiom defining the conditional is presented below. Let X be a subset of *Conf*, α an object which is to execute the conditional, β a boolean expression, s1 and s2 statements. Also, let *err* represent an erroneous condition.

$< X \cup \{< \alpha, \text{if } \beta \text{ then s1 else s2} >\}, \sigma, \tau, \text{u}> \longrightarrow$

$$\begin{cases} < X \cup \{< \alpha, s1 >\}, \sigma, \tau, u > & \text{if } \beta = \text{true} \\ < X \cup \{< \alpha, s2 >\}, \sigma, \tau, u > & \text{if } \beta = \text{false} \\ err & \textit{otherwise} \end{cases}$$

The following rule describes the first action taken to effect the sending of a message to an object represented by the expression 'e' requesting an execution of the specified method m with parameters 'param'. The first action is to evaluate 'e' to identify the object (say e') to which the message is to be sent.

$$\frac{< \text{X} \cup \{< \alpha, \text{e}>\}, \sigma, \tau, \text{u}> \longrightarrow < \text{X'} \cup \{< \alpha, \text{e'}>\}, \sigma', \tau\text{'}, \text{u}>}{< \text{X} \cup \{< \alpha, \text{e! m(param)}>\}, \sigma, \tau, \text{u}> \longrightarrow < \text{X'} \cup \{< \alpha, \text{e'! m(param)}>\}, \sigma', \tau\text{'}, \text{u}>}$$

The rules to actually send/receive are more complex and are not discussed here. The reader is referred to [5] for further details.

The semantics as defined is cumbersome, as the type and the set of definitions are present in all transition rules, even though only a few constructs (like creating a new object/method) will alter these. These constructs are predominantly static. It would have been more elegant to define a compilation phase capturing such information instead of maintaining it in the operational semantics.

The key point in the semantics is the inherent parallelism in the semantics. As the subset of $\mathcal{P}_{\text{fin}}$ can be written in different ways (i.e. X ∪ {< α,s>}, for different α's and s with the statements to be executed by other objects represented by X), all the statments in the subset can be executed in parallel. Abstractly, there is no difference in the semantic style described here and the Plotkin style. Our brief look at it only shows that transition systems can also be used to describe object oriented languages.

3.6 ESTEREL Semantics

In this section we examine closely the semantics of ESTEREL [10]. The syntax of the language was outlined in section 2.5. The language designers have proposed the following three semantics for ESTEREL

- Static semantics to check that programs do not have any temporal paradoxes.
- Behavioral semantics (which are a kind of denotational semantics) to defines completely the temporal behavior but does not worry about effectiveness.
- Computational semantics (or operational semantics) which is more effective than behavioral semantics defining how the execution of a program proceeds.

All three semantics are based on structural conditional rewrite rules defining a transition system. But unlike the other transition systems, the rules are not defined as to form a relation over a set of configurations. That is, it is not essential that the entities on the left and the right of $\rightarrow$ which are enclosed within '< >' are necessarily from the same set.

The transition rules defining the static semantics have the form: $<i,D> \xrightarrow{b,L} <G,D'>$, where i is an instruction (statement), D is a set of signals which determines the execution of i, b a boolean condition indicating whether i may terminate, L the set of exits (tag labels) that i may execute, G is a dependency graph of signals represented as a set of pairs (s1,s2) implying that signal s2 depends on signal s1 and D' the set of signals after executing i. Based on the above definition a program is said to be statically correct, if an only if there exists a transition of the form $<P,\emptyset> \xrightarrow{b,L} <G,D'>$, were G is acyclic. In other words there is no cyclic dependency of signals.

The behavioral semantics defines the history transformation associated with a program. The general idea is to consider the meaning of a program as a transformation under an input event set to a new program, emitting signals in the process.

The transition rules related to the behavioral semantics have the following form. $<i, \sigma, E> \xrightarrow{b,L} <i', \sigma', E'>$, where i is an instruction, σ is a memory state defining the current association of values with variables, E is the input event, b and L as in the static case, i' is the reconfiguration of i (i.e., the new instruction to execute), σ' the new memory state and E' the event output by i.

As the behavioral semantics is not necessarily 'effective', the behavioral semantics are refined (using the static semantics) to yield an interpretation system (or computational semantics). The basic structure of the transition rules for the computational semantics is identical to the behavioral semantics. It is augmented with the graph G (introduced in the static semantics) and an integer n. The integer n indicates the current level of execution with respect to the graph G. The use of G and n will become clear from the example described below.

We present the static, behavioral and computational semantics for the *upto* statement in order. Recall from chapter 2 that the *upto* instruction defines the termination condition for a body in terms of when signals occur. For example, *do* B *upto* S(X) executes B. Whenever the signal S occurs the execution of the *upto* block is terminated and the value associated with the signal is stored in X and the execution continues. The transition rules are:

$$\frac{<i, D \cup \{s\}> \xrightarrow{b,L} <G,D'>}{<\text{do i upto s(x)},D> \xrightarrow{true,L} <G,D'>}$$

$$\frac{s \in E,\ E(s) = v}{<\text{do i upto s(x)},\sigma,E> \xrightarrow{true,\phi} <null,\sigma\,[x \leftarrow v'],\emptyset >}$$

$$\frac{level(s) \leq n,\ s \in E,\ E(s) = v}{<\text{do i upto s(x)},\sigma,E> \xrightarrow{true,\phi,G,n} <null,\sigma[x \leftarrow v'],\emptyset >}$$

The static semantics of 'upto' requires augmentation of the set of signals as specified by the body. The behavioral semantics states that if a signal is present, the associated value is bound to the variable and the upto statement terminates with no signal generation. The computational semantics is a refinement of the behavioral semantics and requires that if s is available at a level less than n, it will be handled within n 'steps' and the value will be bound to x.

Discussion

The advantage of using three types of semantics is that there is a separation of concerns. It enables one to choose a subset that is relevant to one's need. The compiler writer may use the static semantics to perform compile time checks, while the runtime system designer may use the computational semantics. However the presence of multiple semantics could cause difficulty in generating a uniform picture. Care must be taken so that the three semantics when combined do not lead to paradoxes.

In our opinion, the approach of defining semantics which describes the computational semantics and derive the behavioral semantics by defining appropriate equivalences appears to be better as there is only one semantics on which the rest of the work is built.

3.7 Summary

While, each of the techniques described above has its advantages, none of the operational styles explicitly support typing concepts like polymorphism and abstract data types. Another important drawback is that none of the semantic techniques distinguish distributed programs from concurrent programs.

In thesis we remedy both these shortcomings. The semantics for ARL is based on the concept of dynamic algebras [36]. A brief description of dynamic algebra and our justification for this choice is presented chapter 5. Following the introduction, the semantics of ARL is explained in detail. Before doing so, we present the syntactic aspects of the language in the next chapter.

CHAPTER 4

LANGUAGE: SYNTAX

Having discussed our goals and reviewed the literature to identify the various options available to a language designer, we explain in detail the syntactic elements of our language ARL (A Real-time Language.) Recall that the intended domain of application of ARL is distributed real-time system prototyping. Therefore, the emphasis is on ease of programming. In this chapter the syntax for all the language constructs is explained with the meaning being explained with an appeal to intuition. The precise or formal meaning of these constructs is explained in chapter 5.

Overview

In general, an ARL program consists of a set of modules or compilation units, clock units, type units, event type units and interconnections between them. Each module will usually contain local type definitions, subprograms, local event types declaration, event handlers and timing requirements. Clock units contain only clock definitions, while type units contain user defined types and event units contain event type definitions.

In this chapter, the facilities for types and data structure definition are explained first. Being a distributed real-time language, communication using events, time using logical clocks and timing aspects of the language follow the exposition on types. The description of event generation and handling precedes that of subprogram definition and call, as the effects of the handling of events on subprogram calls has to be understood. Definition of the various types of subprograms and handling of subprogram calls will be discussed after the discussion on events and temporal specifications. Finally, the structuring mechanism of modules is described. We consider both the semicolon and the end of line as a statement terminator. Hence the semicolon can be omitted when a statement is ended with a newline. This is only to ease development of programs. In this chapter most of the syntax is explained using examples and skeletal BNF rules. The lexical elements (terminal elements) in the grammar are represented in the bold face. Zero or more elements are captured within [ ], while | represents options for the given non-terminal. Also we assume that id represents an identifier. The BNF grammar for all the principal elements of the language can be found in appendix A.

4.1 Types in ARL

As in many programming languages, assigning a type to objects is central to data security in ARL. Two common techniques of assigning types to objects are 1) explicit typing of objects by the programmer and 2) system inferring a type (or a set of types) from the usage of an object. The first scheme is used in programming languages like Pascal, Ada etc., while the second scheme is used in programming languages like Miranda. In a language whose primary field of use is rapid prototyping, terseness and flexibility for exploratory programming are key issues. In such situations it is unreasonable to require the programmer to specify types for all objects. But on the other hand the programmer should also be able to specify the type if so desired. Hence the language supports both the inferred and explicit techniques for assigning types to objects.

4.1.1 Type Schemes

In the current version of the language a type scheme derived from the Miranda [88] type scheme is supported. However any other type scheme more powerful than this that can be *implemented* and is consistent with our goals can be used. For a list of possible type schemes see [15], where a number of possible type schemes are described. Also [29] discuss type inferencing in the presence of sub-types. The usefulness of these type schemes has yet to be ascertained and is not supported in ARL. Note that a selection from a wide selection can be made as ARL is to be used primarily as a prototyping language. So a slight sacrifice in runtime efficiency can be made for a gain in expressiveness of the language. We discuss a particular extension to the Miranda style type scheme which is supported by ARL. The derived scheme incorporates dynamic type checking along with static type checking.

One of the main drawbacks of Miranda like type schemes is that the type assignment has to be successful at compile time. Runtime checks for type correctness are needed only for explicitly defined variant types. Such a type scheme is called a *static* scheme. Heterogeneous types, like a sequence of either *integers* or *boolean* cannot be assigned to objects unless a variant type is explicitly defined and pattern matching used to identify the sub-cases. Consider the sequence [2, true, 3]. A homogeneous type system *cannot* assign a type it and indicates a type error. In order to help the type system the user has to define a variant record as shown below.

Hetero :: Integer_field *integer* | Boolean_field *boolean*

[(Integer_field 2), (Boolean_field true), (Integer_field 3)] is an instance of Hetero. The definition of variant types requires additional effort on the part of the programmer and is not in keeping with our goal of supporting prototyping. So, we select a particular extension to the Miranda type scheme.

[85] describes a type scheme where type inferencing in the presence of heterogeneous types is supported. It retains the advantage of static typing of not having to use runtime checks to verify type correctness. Only the operations on objects which could not be typed statically require runtime checks. This strategy is a middle ground between the approaches taken by Lisp where objects are not statically typed and all type checking is done at run-time and Miranda where all type inferencing and checking is done at compile time.

Before explaining the type inferencing algorithm used by Miranda and the technique to support incomplete types, the concept and use of polymorphic types is discussed.

4.1.1.1 Polymorphism

One of the main contributions of languages like Miranda to functional programming is the concept of *parametric polymorphism* or specifically the concept of a most general type. When polymorphism is coupled with type inferencing, the language can appear to be untyped without losing the advantages of typed programs. The implementation assigns the most general type possible to all objects and instantiates routines for all the appropriate types. This is an improvement over *generic* types provided by Ada. The programmer need not declare an object to be generic as the system treats an object to be generic if its inferred type is polymorphic. It is also possible to explicitly declare a polymorphic type or function.

In the language, one can indicate polymorphic types using the symbol *, ** etc. For example, 'Tree * :: NILT | NODE * Tree Tree', defines a polymorphic binary tree called Tree. The '*' after NODE indicates that the data item associated with NODE is polymorphic. One can derive a binary tree of integers, or a tree of characters from this definition. The above notation will become clear in a later section. Functions operating on polymorphic types are said to be polymorphic functions.

4.1.1.2 Type Inferencing

Described below is an inferencing procedure for expressions involving only complete types based on the Damas Milner type scheme for functional languages [19]. This procedure has been extended in [85] to support inferencing of heterogeneous types. The inferencing procedure assumes that a function is type correct unless it can prove otherwise. It starts out with a set of assumptions and a set of axioms relating the types of syntactic terms. These axioms are written as antecedent followed by the consequent. For example, $\frac{A \vdash e : \sigma}{A \vdash (\text{let } x := e\ ) : \sigma}$ where $A \vdash e{:}\sigma$ is to be interpreted as: the assignment of type σ to e is consistent with assumptions A. The axiom is to be read as 'if the typing of e as σ is consistent with A, the type assignment of σ to (let x := e), which introduces a new variable x and initializes it to e, is also consistent with A.'

Using similar rules, the type of the function can be inferred. The main work horse in type inferencing is unification. The technique is explained using an example. Define a higher order looping function 'until' in Ada-like syntax as

```
function until (final, trans, state) is
begin
  if final(state)
    then return state
  else
    until( final, trans, (trans state))
end
```

The above function can be written in the equational form as follows

```
until final trans state = state, final state
                        = until final trans (trans state), otherwise
```

The type assignment procedure starts out by assigning a type variable to the arguments and to the result of the function. A type variable represents the set of all possible types. The type assignment to arguments and the result fixes the type of the function. For the sake of argument, let final be of type α, trans of type β, state of type γ and the result of type δ. The type of the value returned by 'until' is of the same type as 'state' viz., γ as indicated by the first equation, thus unifying γ and δ. The right hand side of the first equation requires final to return a boolean type with an argument of type γ. Therefore α is unified to ($\gamma \rightarrow$ boolean). The second equation requires the type of (trans state) to be identical to state. Unification requires β to be identical to $\gamma \rightarrow \gamma$. No other type inference can be made using the above equations. Therefore the function until is typed $(\gamma \rightarrow \text{boolean}) \rightarrow (\gamma \rightarrow \gamma) \rightarrow \gamma \rightarrow \gamma$. As there is a type variable in the final type assignment, the function is polymorphic. This is because the exact nature of the state is irrelevant to the definition of until. We replace the type variables by *,** etc. and the type of until is represented as (* → boolean) → (* → *) → * → *

A similar procedure when applied to the function foldr, which is defined below, will assign the type
(* → ** → **) → ** → [*] → **. Here the function is polymorphic in two variables denoted by * and **

```
foldr op r = f
        where
        f [ ] = r
        f(a:x) = op a (f x)
```

Based on foldr, define a function concat = foldr (++) [ ]. Assume, type of (++) to be [*] → [*] → [*]. Unification of the types assigned to ++ and foldr yields the type of concat as [[*]] → [*], which is a polymorphic in a single variable. Note that concat is a function which concatenates a sequence of sequences into a single sequence.

4.1.1.3 Heterogeneous Types

Having introduced the type scheme supported by Miranda, we elaborate on the work described in [85]. Recall that the reason for wanting heterogeneous, or incomplete, types, is to support mixed types without having to define variant types. A plausible technique to type heterogeneous structures is to consider the union type of all the components. The advantage of such a scheme is that certain errors can be detected. For example, if an object can either be an integer or a character, an operation requiring a boolean type is inappropriate on the object. However, this solution does not handle all expressions. For example, no *finite* number of unions can type the expression (list = cons(1,(map(λx.[x])list))), which is the infinite sequence [1,[1],[[1]] ...]. The type of this infinite sequence is [Integer, [Integer], [[Integer]] ...]. These infinite sequences can arise in our language as we support lazy evaluation. The programmer can specify the infinite list which will be unfolded at execution time only as much as necessary. But the type system is required to assign a consistent type to these objects as there is no a-priori bound on the number of unfoldings. As using unions does not solve the entire problem, a single universal sum type, denoted by Ω is used. This simplifies the implementation as the permissible sum of types need not be remembered.

Axioms to infer types with the ability to detect places that require dynamic checks along with static type checking are developed in [85]. We do not discuss all the axioms, but present an interesting example. The key feature is the definition of an approximation relation $\sqsubseteq$, which indicates a hierarchy of information. For example, $[\Omega] \sqsubseteq$ [int] implies that Ω conveys less information than integer.

Let C denote a set of constraints and A a set of assumptions. Let C$\Vdash$C' denote that any model for C is also a model for C' or in other words C' is consistent with C. The axiom which governs function application is explained below. Let C,A$\vdash$e:τ mean that using constraints C and assumptions A, the type for the expression 'e' can be inferred to be 'τ'.

Axiom 4.1 $$\frac{C,A \vdash e{:}\ \tau' \ \& \ C,A \vdash e'{:}\tau'' \ \& \ C \Vdash (\tau'' \rightarrow \tau) \sqsubseteq \tau'}{C,A \vdash e(e'){:}\ \tau}$$

The above axiom states that if e is typed as τ', e' is typed as τ" and given constraints C and that (τ" $\rightarrow \tau$) $\sqsubseteq \tau$' is consistent with C, the expression e(e') (the result of applying e to e') can be assigned type τ.

For the sake of comparison, the inference rule without incomplete types is

$$\frac{A \vdash e{:}\ \tau' \rightarrow \tau \ \& \ A \vdash e'{:}\tau'}{A \vdash e(e'){:}\ \tau}$$

The static type scheme does not have any consistency assumptions as it does not accept any unexpected type. Hence there is no notion of approximation to be represented by a partial order. The algorithm for heterogeneous type inferencing uses a number of similar axioms and is composed of three distinct phases. We present only an outline here and do not discuss the details. The reader is referred to [85] for more details.

- Find a most general typing for the expression along with a set of verification conditions which must hold for typing to be correct.
- Check for consistency.
- Find a most general solution in the form of a substitution and apply it to the set of assumptions and the general type found.

Having discussed two related type schemes, we present constructs to declare types in ARL.

4.1.2 Type Definition & Operations

Even though type inferencing is supported, the user needs to be encouraged to specify the type when possible. The use of types leads to better readability of programs and can help detect errors. So a number of type definition techniques are supported. To facilitate type definition, the language makes available the predefined types: *integer*, *real*, *boolean* and *character*.

The user can define other types using these predefined types or any of the already declared types. The kinds of types supported include ***tuples***, ***constructors***, ***sequences*** and ***abstract data types***. ***Tuples*** are simple records while ***constructors*** can be used to define variant records or enumerated types. ***Sequences*** are similar to arrays and are usually unconstrained. ***Abstract data types*** permit the specification of types without revealing its internal details. The general form of a type declaration in BNF is

```
type_def :: id :: definition
definition :: tuple_def | constructor_def | sequence_def
```

4.1.2.1 Tuples

A tuple, also known as a record or a product type, is a finite ordered set of elements with each element belonging to an already declared type. That is, the domain of the defined type is the cartesian product of the individual domains. Two techniques to define tuple types are supported. In the first technique, the individual fields are not explicitly named. Pattern matching is used to identify the fields of the type. For example name_rec :: (string,integer) defines name_rec as a 2-tuple type with the first field of type string and the second of type integer. Pattern matching on tuples is specified by a sequence of identifiers within parenthesis. For example, if (x,y) is an object of type name_rec, the type of x is inferred/required to be string and that of y integer.

The language does not provide predefined extractor functions, as it not possible to write a general function which can operate on a general tuple type. The problem arises as one does not know the number of fields in a tuple. However for a tuple of known arity (number of fields), polymorphic extract functions can be defined. For instance, extractor functions of a three tuple can be defined as:

```
first_field :: (*,**,***) → *
second_field :: (*,**,***) → **
third_field :: (*,**,***) → ***
```

An alternate definition of a tuple is to identify each field by a name and specify the type. For example, name_rec can also be defined as

```
type name_rec is record
        name : string;
        soc_no : integer;
end record;
```

In this case, the names of the field can be used as extractor functions. For example, if 'ob' is an object of type 'name_rec', 'ob.name' is of type string and returns the string field of 'ob', while 'ob.soc_no' returns the integer field. The BNF grammar for both the tuple definition style is

```
tuple_def :: id :: ( id [ , id ] )
tuple_def :: type id is record fields [ fields ] end record ;
fields :: id bf : id terminator
```

4.1.2.2 Constructors

Constructors are tags which can be used to define enumerated types and variant records (union types). A constructor based type consists of a series of elements with each element composed of two fields. The first field is a distinct name called the name of the constructor, while the second is a sequence of type names. The name fields of the constructor are to be treated as the tag values for a variant record and have to be distinct for the purposes of type inferencing. A collection of constructor names without any subsequent type field defines an enumerated type. Examples of types defined using constructors are:

```
tree :: NILT | NODE integer tree tree
working_Day :: Mon | Tue | Wed | Thu | Fri
```

In the above examples, working_day is an enumerated type, while tree is a variant record. In the type tree, if the tag value is NILT the tree has no data field, while a tag value of NODE indicates that the value field is a 3 tuple consisting of an integer and two trees. Note that ARL (like Miranda) requires the name of the constructor starts with a capital letter. This information is used in type inferencing.

Pattern matching has to be used to identify the variant case. If an object x equals NILT, the type of x is Tree and its tag is NILT and has no other field, while if x equals (NODE n y z), x is of type tree with tag NODE and a value field consisting of an integer n and two subtrees y and z. Type definitions using constructors is governed by the following BNF formula.

```
constructor_def :: id :: constructor [ | constructor ]
constructor :: cap_id [ id ]
```

The effect of constructors can also be achieved by explicitly defining a variant record in the Ada style. For example, the type Tree declared above can be defined in Ada as

```
type tag is (NILT, NODE)
type tree(typ : tag) is record
    case typ is
        when NILT => null;
        when NODE =>
            value : integer;
            left,right : Tree;
    end case;
end record;
```

The BNF for variant record case is

```
constructor_def :: type id ( id : id ) is record fields [ fields ] case_opt
                                   end record
case_opt :: case id is case_field [ case_field ] end case
case_field :: when id => fields [ fields ]
```

While the first style results is a succinct definition, it forces one to use pattern matching to identify the individual fields of a variable. In the second case one can use the field names like 'left' etc., to identify a specific field.

4.1.2.3 Sequences

Sequences are unbounded arrays with *all* elements of the sequence having the same type. Mathematically speaking, sequences are well ordered multisets. As usual, two schemes to define sequences are provided. The first scheme borrowed from Miranda while the second scheme is from Ada.

In Miranda, '[' ']' are used to denote sequences. For example, string :: [char] defines a type string as a sequence of characters. The lower index is implicitly assumed to be zero and there is no limit on the upper index. Multidimensional arrays are constructed by using nested sequences. For example, multi_d ::[[integer]] defines a multi-dimensional array (sequences of sequences) of integers. [[1,2][3,4,5][6]] is an instance of multi_d. Notice, that is possible to have a multi-dimensional array with each row (or columns) a having different size. The first of the above examples of 'string' can be defined in the Ada style as:

type string **is array** (<natural>)

However, we retain the restrictions specified by Ada, and require an object of the above type to have the bounds specified during declaration. This in our opinion will result in a more efficient implementation and should be use when efficiency is desired. However, this is less flexible than sequences in Miranda. It is also not possible to have rows (columns) of different sizes. That declaration of multidimensional arrays is a straight forward extension to the above. The BNF grammar for sequences is

```
sequence_def :: id [ id ]
                type id is array ( range )
```

The various operations that can be performed on sequences are defined. Patterns representing sequences are (a:x) where a is the first element in the sequence and x is the rest of the sequence, which could be null. This is useful in representing non-null sequences as parameters to subprograms. ':' can be used more than once in a pattern. a:b:c:x indicates a list of at least three elements with the first aliased to a, the second aliased to b and the third aliased to c. The ':' operator can also be used to add an element to the head of a sequence. For example, return 1:[2 3 4] returns the sequence [1 2 3 4]. Concatenation of two sequences is defined by the '++' operator. For example, [1 2 3] ++ [3 4] = [1 2 3 3 4]

A sequence subtract operator −− is also defined. It converts the arguments to multi-sets and performs a set difference. The ordering in the resultant sequence is the ordering induced by the first sequence. For example, [1 2 3] −− [2 3 5] = [1] and [1 2 3 4 5 2 4] −− [3 2 5] = [1 4 2 4]. The −− operator can formally be defined as

```
x −− [ ] = x
x −− (b:y) = (remove b x) −− y
remove b [ ] = [ ]
remove b (a:x) = x, a=b
               = a:(remove b x), otherwise
```

An arithmetic series (a sequence of integers) can be abbreviated by '..'. [a .. b] represents a list of numbers from a to b inclusive with an increment of 1 (the default increment.) [a,b .. c] is an arithmetic series with first number a, second number b and the last member not exceeding c if b-a is positive and not less than c if b-a is negative. Infinite sequences are represented by omitting the last element. [1 ..] is the sequence of positive integers, while [2, 4, 6, ..] is a sequence of even integers.

Z-F (Zermelo-Fraenkel) expressions generate new sequences from existing sequences. A Z-F expression consists of an expression and a sequence of qualifiers. A qualifier is of the form 'pattern <- sequence of expressions.' A qualifier generates a list of values to be passed to the expression. The result of evaluating the expression is added to the sequence. For example, factors n = [r | r <- [1 .. n div 2] ; n mod r = 0] defines a sequence containing the factors of the number n. The Z-F expression used above is to be interpreted as: for every r in the sequence [1 .. n div 2], if n mod r equals 0 add it to the resulting sequence. In the above example, 'r <- [1 .. n div 2]; n mod r' is the qualifier and r is the expression. Also, r to the left of the <- is the pattern and [1 .. n div 2] and n mod r = 0 the sequence of expressions.

Z-F expressions can be used to construct lists from recurrence relations by permitting the variable(s) on the left hand side of the <- in the qualifiers. [x | x <- a , (f x)] is the infinite sequence [a, f a, f(f a) ...] The Z-F expression is to be interpreted as select x from a and f(x) and add it to the sequence.

The use of multiple generators is to be interpreted like nested *loops*. For example the value of [f x y | x,y <- [1 2]] is [f 1 1, f 1 2, f 2 1, f 2 2] An equivalent Ada like code fragment is

```
seq := [ ]
for x in 1 .. 2 loop
    for y in 1 ..2 loop
        seq := seq ++ [f(x y)]
    end loop
end loop
return seq
```

This definition of nesting is not satisfactory for infinite sets, as certain values may never be reached in an enumeration. To provide a satisfactory nesting rule, the Cantor diagonalization algorithm is used [26]. The scoping rules in the new looping construct are the same as that in the original loop. To denote the diagonalization '//' is used instead of |. For example:
[f x y // x,y <- [1 .. 3]] = [f 1 1, f 1 2, f 2 1, f 1 3, f 2 2, f 3 1, f 2 3, f 3 2, f 3 3]

Using Z-F expressions and infinite integer sequences other infinite sequences can be constructed. Z-F expressions are important not only for their terseness but also because their proper usage can eliminate the use of recursion in functional programs. For example, the cartesian product of two lists x and y can be specified as

    cart_product x y = [(a ,b) | a <- x, b <-y].

The general form of a Z-F expression is governed by the following BNF rules

```
zf_expr :: [ expr iterator qualifier [ qualifier ] ]
        :: [ generator | qualifier [ qualifier ] ]
iterator :: | | //
generator :: pattern <- expr [ , expr ]
qualifier :: expr | generator
```

4.1.2.4 Algebraic data types

The usefulness of algebraic or abstract data types is a well studied topic and we do not discuss their advantages [38, 100]. Abstract data types allow the user to define a type together with some operations on it without exposing the internal details of the type. We use the universal example of a stack as an example of abstract data types.

```
abstype stack *
with
   new_stack :: stack *
   is_empty :: stack * -> boolean
   pop :: stack * -> stack *
   top :: stack * -> *
   push :: stack * -> * -> stack *
implement
   stack * == [*]
   new_stack = [ ]
   pop (a:x) = x
   top (a:x) = a
   push x a = (a:x)
```

The keyword **abstype** names the type being described, while **with** introduces the operations on objects of the defined type. All definitions following the **implement** are hidden from the users of the abstract type. A '==' statement after **implement** defines the concrete type. In the above example the stack is represented as a sequence. The operators are then defined on the concrete type.

One could also define an abstract data type in terms of the properties it satisfies and operations to ensure the adherence to the properties. Once again, our example is drawn from Miranda. Define a type 'rational', the rational numbers so that they are always represented in their lowest terms. A constructor RATIO is defined which has two integer fields. The first integer represents the numerator, while the second field represents the denominator.

```
rational:: RATIO integer integer
   RATIO p q
   => RATIO (-p) (-q), q <0
   => RATIO 0 1 , (p = 0) & (q ~= 1)
   => RATIO (p/h) (q/h), (p ~= 0) & (h > 1)
     where
       h = hcf (abs p) q
       hcf a b = hcf b a, a <b
```

```
= hcf (a **mod** b) b, a **mod** b>0
= b, **otherwise**
```

Such definitions are composed of the type definition (constructors in the above example), followed by 'rules' about the definition. Note that these rules could introduce local definitions using the **where** clause as shown above.

The BNF grammar for algebraic types is as follows. Towards it we define string_of_stars to represent *, **, *** etc.

```
abstype :: abstype id [ poly ] with signature
abstype :: constructor_def [ law ] local_declaration
poly :: string_of_stars
signature :: [absfunc] implement absdefin
absfunc :: id :: type_indication
absdefin :: subprogram_decl
type_indication :: id [ poly ]| type_indication -> id [ poly ]
law :: pattern => expr [ , expr ]
local_declaration :: where [ declaration ]
```

This concludes the discussion of data types in the language. The complete BNF grammar can be found in appendix A. In the next section, we discuss how time and distribution can be specified in ARL.

4.2 A Concept of Time

In this section, we show how logical clocks are used to introduce a definition of time. As discussed in chapter 1 we permit multiple clock definitions in order to be able to denote distribution. Note that when we say denote distribution we do not mean the specification of what goes where, nor do we mean relative remoteness. The mapping of code onto processors or indication of relative remoteness is an issue which is best left to an implementor [17]. The reasons for this are discussed in detail in section 4.7.2. What we do mean by denoting distribution is that the meaning of a program is indexed by the mapping of clocks to other syntactic issues. This is elaborated further in section 5.8.

Having permitted the definition of multiple clocks, there needs to be constructs which help maintain a global notion of time within certain error margins. In this section we also show how to synchronize the various clock definitions.

It is essential to have a concept of time in order to be able to specify any real time system be it hard real-time or soft real-time [82]. One way of characterizing time is to do away with an absolute time frame and denote the progress of time by defining points where a relevant 'state change' occurs. The interval between consecutive points can be assigned a metric to measure the elapsed time. But this by itself is insufficient as it does not give a precise sense of time. It is impossible to *guarantee exactly* when relevant state changes occur in all executions. That is, one cannot be sure that a change will occur exactly t units of time after another. An added deficiency of this technique is that it does not permit the characterization of static periods (finite/infinite) and does not have an absolute control flow independent notion of time.

An alternate technique to use one of the time standards defined in [51]. [51] classifies time into the following time standards: 1) Time based on *astronomical* observations, 2) *External Physical* time like UTC, 3) *Physical* clock like an oscillator, 4) *Internal Physical* which approximates external physical time 5) *Local* time is the time of the local physical clock in a distributed environment.

We introduce a slightly different concept of logical clock, i.e., a variable whose value is updated at a fixed period. We *do not* define a mapping between a logical clock and any of the standards outlined above. This is because we believe that the mapping will depend on the application. This lack of mapping is not a serious limitation as ARL is to used primarily for prototyping systems. In other words, the only feature to incorporate time in the language is to use explicit defined logical clock(s) and specify timing with respect to this/these logical clock(s). This technique is used in ARL, as it results in a more precise definition of time than an event based time definition. It also allows us to characterize static periods (periods of no apparent change) denoted by the passage in time signified by an increase in the value of the clock.

To model real systems and to simplify the semantic model, the language supports only a discrete notion of time. We do not consider this a serious limitation as programs generally do not use or depend on the denseness of time. The discreteness also permits an implementation to directly map logical clock(s) to physical clock(s).

The necessary syntax to define discrete clocks is discussed. As time 'changes by itself' and not due to any program dependent execution, we use what we call an **auto** procedure. In general, an **auto** procedure is any 'segment of code which is executed periodically.' As the value associated with a clock is to be updated periodically irrespective of what a program achieves clocks are a specific type of **auto** procedures. Clock definitions come in two flavours. The first defines a single clock, while the second defines an array of clocks. The use of the array definitions will become clear when when the units of distribution is discussed in section 4.7.2. The array definitions is only a syntactic convenience and the clocks are independent except for clock synchronization. The BNF grammar for the definition of clocks is

clock_def :: **auto** id param_spec_opt := id + expr **init** expr
param_spec_opt :: id | id mod_param
mod_param :: [mod_seq]
mod_seq :: id : expr .. expr [, mod_seq]

An example of a clock definition is: **auto** id := id + expr **init** n where expr is a positive, integer valued expression determining the increment of the clock and n the initial value of the clock. The value domain of the clock id by itself is the set { n, n+expr, n+2*expr .. } well ordered by the natural $\leq$ relation. An example of an array definition is **auto** clks[index: 1.. 10] := clks + f(index) **init** g(index). It defines 10 clocks indexed by 'index' with initial values given by g(index) and increment f(index).

While the meaning of the initialization field is obvious, the meaning of the 'increment' field is not. It is reasonable to interpret the 'increment' field as a measure of either 1) the frequency or 2) as the resolution of the clock. This will be discussed briefly in sub-section 4.2.2. However, the *absolute* frequency of increment or resolution of the clock is not defined by the language and is left to the implementation.

4.2.1 Distribution

In this section we show how the definition of time can be used to denote distribution. It is reasonable to assume that each machine in a distributed environment has a local sense of time. It is widely accepted that a single global sense of time is not sufficient to characterize a distributed system. We assume that time at each site is maintained by a local physical clock. A distributed system can be represented by a program involving multiple clocks. Our technique of unifying the concept of time and distribution is unique in the area of programming languages. Very few languages have an absolute time frame in the language itself and none of these use multiple time definitions to characterize distribution. This unification is especially significant for prototyping systems, as it has enabled the introduction of multiple concepts at the program level with a single syntactic entity. In other words, by using a single concept of clocks, the programmer is able to specify both the notion of time and distribution. The system specifier can use multiple auto functions, to define a multi-clock and hence use an important aspect of distributed systems to denote distribution.

We reiterate that by this we have not defined the units of distribution nor have we discussed how distribution can be achieved. We have only said what in ARL characterizes distribution. The potential choice in the unit of distribution is discussed in section 4.7.2.

The price to be paid for permitting multiple clocks is that clocks are no longer monotonic. This is because a uniform notion of time is essential and the programmer has to deal with clock synchronization. Also reading of a 'remote' clock is not instantaneous and hence an interval notion of time is necessary. This is discussed in detail in the next chapter on semantics.

4.2.2 Interpretation of Increment Field

As mentioned above, there are two possible interpretations to the increment field in a clock namely, frequency or resolution. In the frequency interpretation, the **incr** field represents a relative frequency of the clock, while in the resolution of the clock, the **incr** field represents a relative resolution of the clock.

Towards the explanation of the frequency interpretation, assume the existence of a uniformly varying and continuously updated clock T. Also assume that all logical clocks defined in the program behave perfectly such that the time represented by them is incremented at fixed intervals. Also assume that the clock values are incremented by one. Consider two clocks c1 and c2 such that the **incr** field specified for c1 is k1 and for c2 is k2. The intuitive relationship between the speeds of the clocks is defined more precisely as follows. Define o(C) to be the time elapsed with respect to T for C to be incremented by 1. Using this definition, the relation between c1 cand c2 can be defined as: o(c1)/o(c2) is equal to k2/k1.

In other words, the above condition states that the actual rate of increment of the clocks are in the same ratio as their specified increments. Consider the following example. Let a clock c1 have an increment of 2 and a clock c2 have an increment of 5. In one time unit with respect to an appropriate clock, the clock c1 is incremented twice while clock c2 is incremented five times. Note that if the frequency interpretation is adopted, all mapping of these logical clocks to physical clocks must respect the relative speed requirement. For example, if clock c1 has twice the speed of clock c2, the physical clock corresponding to c1 must have twice the speed of the physical clock corresponding to c2.

In the resolution interpretation, the increment represents the actual time that has elapsed before consecutive readings of the clock yields a non-zero value. Hence the minimum non-zero time difference between two consecutive calls to the read the time will be in proportion to the specified increment. Let Ω_X be the actual resolution of clock X. The relation between c1 and c2 is Ω_{c1}/Ω_{c2} is equal to k1/k2.

For the purposes of the current language definition, we prefer the *resolution interpretation*. This is because, we do not distinguish between absolute time (or calendar time) and time read by the clock. If different clocks were of different frequencies, creation of an absolute time is essential. This unnecessarily complicates an already complex issue and we recommend the resolution interpretation.

In summary, the increment field in a clock definition represents a *relative resolution* with respect to other clocks. We also assume that all clocks have the same ideal frequency so that we need not differentiate between absolute time and logical time. That is, ideally all the clocks in the system keep identical time, but with different resolutions.

4.2.3 Clock Synchronization

Experience shows that all hardware clocks drift from their expected values. In systems involving multiple clocks, there is a need to check the drift and keep the clocks related to one another. In other words, the clocks must not be allowed to drift too much from their expected values and must be synchronized with respect to others in the system. [51] defines two levels of clock synchronization. The first is *internal* synchronization which refers to the construction of an approximate global time base among the ensemble of nodes of a distributed system. The other is *external* synchronization which refers to the synchronization of the approximate global time base with the external physical time, in order to generate an internal physical time. Note that even clock synchronization does not guarantee a perfect time base and there always will be some errors. Clock synchronization only keeps these errors within certain bounds. In ARL, synchronization will refer to *internal* synchronization as it does not define a physical time base.

The language must either define a default synchronization interval or must provide techniques to specify synchronization. [80, 51], to cite a few, discuss clock synchronization algorithms. These algorithms are valid only under strict assumptions of knowing the clock drift rate, network delays etc. Availability of information such as the network delay cannot be assumed by the language definition. Different environments will dictate different delays, and it would be inappropriate for the language to define a fixed synchronization interval. The synchronization of clocks will also depend on the resolution and the accuracy of the physical clocks. Therefore we define a construct to specify clock synchronization. The semantics will abstract the meanings thus permitting the use of a class of synchronization algorithm.

Two classes of clock synchronization algorithms are supported by ARL. The first is the synchronization of one clock with respect to another, while the second synchronizes a set of clocks with respect to each other. We discuss them in order.

The general form of the construct to specify the synchronization of one clock with respect to another is

clock_sync :: **auto** id >> expression id := id.

Consider '**auto** ((timer1 >> n) timer2 := timer1)'. It is an example of clock synchronization whose functionality is to synchronize timer2 to timer1 at every n units of time with respect to timer1. The meaning of the statement can be informally described as follows. The clock timer1 sends its current value to timer2 every n units of time as measured by timer1. Timer2 on receipt of the time from timer1 will change its time 'based on the value received'. The reason for not having a direct assignment is to allow for corrections due to message transfer etc. Depending on the characterization of time, the value assigned will either be an integer or an interval. The integer is only an approximate notion of time, while the interval represents the error in the time associated with timer1 as seen from timer2. The details of this is discussed in section 5.3.3.

Note that in the above syntax only one clock is synchronized with respect to another clock. It might be desireable to synchronize a set of clocks with respect to each other. We call a scheme where a set of clocks are synchronized with respect to each other as broadcast synchronization. In order to permit broadcast synchronization as used in [51, 80], the above syntax is generalized. In broadcast synchronization, when each clock detects that a specified interval has elapsed, it broadcasts its time (local time) to other clocks involved in the synchronization. When a clock has received sufficient information from other clocks, it re-adjusts its clock appropriately. The details of such a scheme will depend on the actual algorithm used by the implementation. [80, 51] describe algorithms to effect broadcast synchronization. The grammar governing this syntax is

broadcast_sync :: **auto** (id [, id] : expr)

For example, **auto**(C1,C2, ...,C_n: T), where C1, C2, ... C_n are clocks to be synchronized with respect to each other every T (an integer) units of time denoting the periodicity of synchronization.

In the above section we have defined the constructs to effect clock synchronization. We have not defined how this affects the nature of time. As mentioned earlier, time can be characterized by either an integer or an interval. For example, due to time delays and inaccuracies of the clocks involved, time as an integer may no longer be sufficient. Time may have to be represented as an interval. Such issues are discussed in detail in chapter 5.

4.2.4 Periodic Tasks

The scope of the **auto** and the '>>' statement is generalized to facilitate the definition of periodic tasks. A periodic task, as the name suggests, is a task that is to be performed periodically and ideally should be activated at fixed intervals. The periodicity of such a task is specified with respect to an already defined clock. The BNF rule for a periodic task is

per_tsk_decl :: id_opt **auto** id >> expression subprogram_call
id_opt :: | id :

The following is an example of a periodic task:

name:**auto** (timer1 >> m) (P arg1 .. argk)

The subprogram P with it's k arguments is scheduled for execution every m time units with respect to clock timer1. The arguments are expressions which are evaluated *every time* the periodic task is scheduled for execution. These expressions could refer to mutable objects defined in the compilation unit in which the periodic task is defined. Note that timer1 must be defined as a clock using an **auto** statement. The 'name' field identifies the specified task.

4.2.5 Delay

The language provides two types of **delay** statements. The first is to delay until an absolute time is reached while the other is to delay for a specified durations. As it is impossible to guarantee a precisely delay the effect is to delay *atleast* until the time is reached and *atleast* as long as specified respectively. The language does not place an upper bound on the delay. This is because the upper bound depends on the nature of scheduler, the criticality of the process waiting etc., which are not known. An implementation may choose to delay for an arbitrarily large time as long as it is greater than the specified time. But if the implementation delays for too long the program could generate temporal violations (to be discussed later.) Hence we delegate the building of a robust program to the programmer who should account for bad implementations of delay using other temporal specifications. In other words, upper bounds are to be specified using temporal specifications. The nature of temporal specifications allowed by ARL are discussed in section 4.4.

The syntax of the delay statement is as follows

delay_stat :: **delay** until_opt expr **wrt** clock_spec_opt
clock_spec_opt :: id | id [expr]
until_opt :: | **until**

For example, **delay** 10 **wrt** cl, delays the execution of the current thread for atleast 10 units of time measured with respect to clock cl, while **delay until** 100 **wrt** c2, delays the execution until c2 reads atleast 100.

If the 'delay until' statement is only executed after the specified time has already elapsed, the statement has 'no effect'.

4.3 Events

The rationale for introducing events in the language is two fold. First, events are a convenient tool to represent asynchronous aspects of the system and the environment with which the system interacts, e.g., interrupts. Any relevant asynchronous behavior by the environment in which the program executes, can be modeled by generating an event of the appropriate type. Events can also be treated as units of communication by attaching a value which represents the message with an event.

Communication in a concurrent environment can either be synchronous or asynchronous. In synchronous communication, the sender waits till a receiver is ready to accept the communication. In asynchronous, the sender mails the message and continues executing without waiting for receiver to accept the message. We chose the asynchronous communication model as it is more elementary than the synchronous model. In a real-time environment, the idea of waiting for a process to accept a message seems too restrictive. The delay of one process should not affect the progress of the other process unless it is absolutely necessary. The necessity of this can only be decided by the programmer, due to which the language supports asynchronous communication. The idea of asynchrony can also be used to model exceptions and faults.

We do not wish to restrict the use of events in the language, as it nearly impossible for us to consider all possible situations where an event is essential. For example, we do not suggest in the language any specific exception handling mechanism. The programmer has to choose the best technique for the application and implement it using events. In general, events are to be used to denote significant points during the computation, which include but are not restricted to, interrupts, entering/exiting a function, I/O, exceptions etc.

4.3.1 Event Types

Different events of similar functionality can be grouped to from an event type. In keeping with our philosophy of a typed language, all values which can be associated with an event of a type must belong to a data type. Thus, an event type refers to an identifier called the *event_name* and the set of values associated with the name. An implementation is required to maintain other fields along with the event_name and value. The fields are the originator of the event (name of the encapsulation unit that generated it), the event_name, the value associated with it, time of occurrence and the clock with respect to which time was measured. The reason for this is explained in chapter 5 when the precise semantics of such constructs is developed.

Event declaration comes in two styles. In the first the value associated with events of the event type is not specified while in the second case it is. If the first option is used (i.e., the type specification is omitted) an implementation should be able to infer the type of the value field from the usage. The BNF rule governing an event type declaration is as follows.

event_type_decl :: **event** id [, id] |
                               event id [, id] :: id

A compilation unit has the option of restricting the scope of events. It may choose only to handle or only to generate events of a particular type. The rule to specify such restrictions is

event_restrict :: **input** id [, id] | **output** id [, id]

The key word **input** indicates that the event(s) of the specified type(s) can only be handled by the program unit. That is, events can be 'input' to it by other compilation units. The program unit however *cannot* generate any event of that event type and all statements which instantiate an event are disallowed. The key word **output** on the other hand signifies that the event(s) of the type(s) can only be generated but cannot be handled by the program unit. For example, '**input** E1,E2,E3' specifies that events of type E1, E2 and E3 can only be handled and should not be generated. Similarly, the **output** command requires that the specified events cannot be handled. Therefore no handler involving these event types can be present. For example, '**output** E4,E5' specifies that events of type E4 and E5 cannot be handled but can be generated.

A compilation unit consisting only of event declarations is called an event pool. An event pool can be 'included' by other compilation units, thus facilitating sharing of event types.

4.3.2 Event Values

In order to make the language uniform, one might be tempted to make functions, infinite lists etc. as first class objects [89]. This requires functions, infinite lists etc. to be allowed as event values. However, as pointed on in [89], functions and lazy lists are 'closure' objects and contain embedded references to any item in the heap space of the process that created it. In a distributed setting, the data structures at the various sites will have to be intertwined possibly requiring a global heap. To characterize certain closure objects which may be passed from site to site, the concept of hyperstrict is introduced in [89]. A function is said to be hyperstrict, if all arguments to it are completely defined. In other words, there is no undefined term ($\perp$) anywhere in the object. For example, any function operating on an infinite tree is not hyperstrict as any finite representation of the infinite tree necessarily has a $\perp$ embedded in it. Using this definition it is possible to permit certain types of functions as event values but this only adds to the non-uniformity of the language. We disallow closure objects to be used as event values. We feel that this restriction of only data types is necessary if the language is to be used in the building of realistic distributed real-time systems.

4.3.3 Event Related Statements

Event usage can be divided into two classes: generation and handling. Specific events are generated by the **generate** command. The statement takes as arguments an event name and a value of the appropriate type. The effect of the generation is to broadcast the pair to *all* modules in which the event type is visible for handling. For example, **generate**(e,v) results in the (e,v) along with other information being instantiated on all modules which can handle the event type e. Thus, the execution of the generate command results in the broadcasting of information. The general form of the generate statement in BNF form is

gen_stat :: **generate** (id , expr)

As events were introduced in the model to characterize asynchrony, a declarative statement is a natural way to specify their handling. The definition of these statements (also called causal statements) and its intuitive semantics is described below.

Consider the following example. Assume that input event types e_1,e_2,e_3 ... e_n (not necessarily distinct) and subprograms g_1, g_2, g_3 ... g_n, f (not necessarily distinct) all of which return *boolean* values are defined. An example of a *causal* statement is presented below.

$$g_1(e_1), g_2(e_2), \ldots, g_n(e_n) \textbf{ causes } \mathrm{f}(e_{i1}, e_{i2}, e_{i3}, \ldots e_{ik})$$

Note that the indices $i1$ to ik have to be in the range 1 .. n but need not be completely distinct. The restriction permits only the events on the left hand side of the **causes** to be used on the right hand side. One need not use all the event values and one may use a particular event value more than once.

Towards the meaning of the above statement, define a *cycle* as a sequence of events which cause all the g_i's to execute an arbitrary number of times before forcing f to be executed. When an event of type e_i occurs then g_i, which is a boolean value returning subprogram (side effects are allowed), is invoked and a new thread is said to have been started. If g_i returns **true**, it waits for another event e_i. Otherwise g_i in the current specification is said to become 'passive" and g_i will not handle any further e_i's in the current cycle. When all the g_i's become passive, f with the appropriate arguments is evaluated, and if f returns **true**, all the g_i's become active again and the cycle is repeated. Note that we *do not* require the sequence of events in two *cycles* to be identical. A sequence of events in a cycle change the state of the handlers from 'active' to 'passive' when the right hand side is invoked. If f returns **false**, a predefined event **disaster** is generated after which the g_i's become active. All events which occur during the period when the relevant handler is **passive**, are *queued.* The 'name' field identified the causal statement.

The motivation for the above syntax comes from an attempt to specify fault tolerance. The occurrence of event e_i is an indication of the need for certain checks. If something has gone awry, a corrective action is attempted. The returning of **true** by the g_i is an indication that either nothing was wrong or that the system recovered from the error. A **false** value indicates that the system could not recover from the error but can continue to function without further action. When a number of such errors have occurred, signified by when all g_i's become passive, a subprogram f has to be invoked. The subprogram f can be written to have 'global information' to recover from all the previous errors. The returning of **true** by f indicates that the corrective action was successful. Otherwise some catastrophic situation has arisen, signified by the generation of an event of type **disaster**.

However we do not wish to restrict the use of events to only fault tolerance and they may be used to specify any type of behavior by the programmer. Events essentially indicate a state (or a set of states), not necessarily faulty, of the system where a particular action is necessary. The use of events is only restricted by the semantics of the construct defined. The general form of a causal statement is

```
causal_stat :: handler [ , handler ] causes subprogram_call
handler :: id ( id )
```

A simple example demonstrating the use of events is presented below. More detailed examples are discussed in chapter 6. Consider controlling an object which is oscillating between two points. When a sensor detects that the object has reached the left end point, an event 'left-edge' is generated with the velocity of the object as the value. This should cause the object to start moving in the other direction with an acceleration which depends on the current speed. The behavior of the right end can be stated similarly. In the following program segment, let || represent a comment

```
input left_edge, right_edge || input event restriction
output acceleration || output event restriction

false (left_edge) causes move (left_edge)
|| the event type name can be used to specify the value associated with the event.
false(right_edge) causes move (right_edge)
move x = generate (acceleration, func(x)); return true || generate an event
```

false is a predefined polymorphic function which always returns the boolean **false**. When the object reaches the left end, it generated an event left_edge. **false** is invoked which returns **false** invoking move with the value associated with the event left_edge namely the velocity of the object. The subprogram move generates an event called acceleration with a value depending on the velocity. This event should be detected by a controller to move the object to the right. The event right_edge evokes a similar response from move. The statement generating the event acceleration, returns **true** to prevent the cause of **disaster**.

As ARL is a real-time language, it has constructs to specify timing requirements. This is discussed in section 4.4. As the satisfaction of these requirements is not guaranteed, violations must be signaled. Similarly, if a periodic task cannot be scheduled within the specified period, a scheduling violation must also be signaled. Two more predefined event types provided by the language are: **temporal_violation** and **periodic_violation**. An event of type **temporal_violation** is generated whenever a temporal constraint is violated. The type of the value field of this event type is explained in the section of temporal violations. By then it will be clear what information needs to be attached to this event type for recovery. Whenever it is detected that a periodic task has not been scheduled during a period, an event of type **periodic_violation** is generated. The value associated with it is the name of the periodic task.

4.3.4 Rules for Type Inference

Inference rules to handle event generation and handling are introduced. These inference rules are added to the existing set of inference rules. There are two inference rules for the generate statement. The first rule deals with completely defined types, while the second rule is for partially defined types.

In a generate statement, if the value field is of type τ, the type of the data element associated with the event type (denote it by a function called value_field) should also be τ. If an event type e's data field is assigned types τ_1 and τ_2, (say due to two different generates), any type that has less information than τ_1 and τ_2 is an acceptable type for the value field of e. These are described formally as

$$\frac{A,C \vdash v{:}\tau \;\&\; \textbf{generate}(e,v) \in \text{STATEMENT}}{A,C \vdash \text{value\_field}(e){:}\tau}$$

$$\frac{A,C \vdash \text{value\_field}(e){:}\ \tau_1 \;\&\; A,C\vdash \text{value\_field}(e) : \tau_2 \;\&\; C|\!\vdash \tau \sqsubseteq \tau_1,\ C|\!\vdash \tau \sqsubseteq \tau_2}{A,C \vdash \text{value\_field}(e){:}\ \tau}$$

The causal statement can also be used to infer event types and types of the subprograms handling events. Let g_i be a subprogram handling events of type e_i. The following type inference rules are added.

$$\frac{A,C \vdash g_i{:}\tau \rightarrow \text{boolean}}{A,C \vdash \text{value\_field}(e_i){:}\ \tau}$$

$$\frac{A,C \vdash \text{value\_field}(e_i){:}\ \tau}{A,C \vdash g_i{:}\ \tau \rightarrow \text{boolean}}$$

$$\frac{A,C \vdash g_i{:}\ \tau \rightarrow \text{boolean} \;\&\; A,C \vdash \text{value\_field}(e_i){:}\ \tau' \;\&\; C\ |\!\vdash \tau' \sqsubseteq \tau}{A,C \vdash g_i{:}\ \tau' \rightarrow \text{boolean}}$$

The first rule states that if the argument to g_i is of type τ, the value field of e_i can then be typed τ, while the second rule states that if value field of e_i is typed τ, the subprogram g_i can be typed as $\tau \rightarrow$ boolean. The third rule is to deal with incomplete types. If g_i is typed as $\tau \rightarrow$ boolean, and the value field of e_i is typed as τ' and τ' is less defined than τ, the type of g_i is relaxed to accommodate τ'.

4.3.5 Timing Event Occurrences

As events are the main items in a temporal specification, it is important to define precisely when an event occurs. There are two main strategies in assigning time to events. Events can either be atomic or non-atomic. We discuss the implications of each case.

First consider non-atomic events. For example, communication from one site to another can be said to occur from the time of generation to the time of receipt. One can then define duration of event as a two tuple (t_{start} ,t_{finish}). In a distributed system, this definition is insufficient by itself as the clock with respect to which these times are measured is left undefined. The clock with respect to which time is measured is important as ARL permits the specification of a mapping of clocks to syntactic units. Therefore only a set of such values can characterize an event occurrence. We emphasize that one send and many receives is considered to be a number of single event generations.

Atomic events on the other hand need to have only one component (say the t_{finish} component). In a distributed system, we do need a set of values (one per clock) but do not need a set of tuples. Therefore atomic events are simpler to characterize than non-atomic events.

We examine if non-atomic events add to the expressiveness of the language. The principal use of non-atomic events in other languages like Lustre [16], is to characterize interval due to which the start and end of events were defined. However, in ARL, intervals can be characterized using clock values. Therefore there is no need to differentiate between the starting and ending of events. There is no loss in the expressive capability of the language if events are treated as atomic. Hence to simplify matters event occurrences as defined to be atomic. An event is said to occur when the last instruction to generate it is executed. The last instruction is defined precisely in the chapter on semantics.

4.3.6 Specific Communication

The broadcast communication interpretation for the generation of an event is not expressively complete as it cannot model a channel. An added disadvantage is that broadcast communication is not very efficient. It is possible to increase the expressive power of the language and support specific communication without added syntactic baggage.

To achieve specific communication, the **reply** command is introduced. The behavior of the **reply** command is different from the **generate** command only in a thread activated by an event occurrence, otherwise its semantics are identical to that of **generate**. If the procedure in which the **reply** command is used is in a thread activated due to an event generation, the **reply** statement generates the event such that it is visible only to the module that generated the original event which caused the invocation. For example, let a unit M generate an event which causes procedure p to be executed. Let **reply**(f,v) be executed in p. Instead of the normal broadcast semantics of generate, **reply**(f,v) sends (f,v) only to unit M.

If more than one module responded with a **reply** to the original **generate** command, a set of channels has been established. A single channel can be established by ensuring that the original message is relevant to only one module. The module for which the message is relevant replies, while the other modules ignore the event. This can also be achieved by restricting events using **input** and/or **output**. The BNF grammar for the reply command is

reply_stat :: **reply** (id , expr)

4.3.7 A Note on Parallelism

Different events generated by different(same) modules could arrive at a module in an order such that the second event occurs before the handler for the first event completes its execution. It is conceivable that an implementation could start the execution of the handler for the second event concurrently or in parallel with the first handler. However, we require that at no time do two instances of the same handler execute concurrently with one another. This is to achieve a serializable computation and also keep the semantics of a cycle simple. If not, it would be possible to modify variables in a random fashion leading to non-serializable execution. It would also be possible that, if multiple instances of the same handler were allowed, for a later invocation to finish before the earlier. If the later handler returned **false**, it is possible to activate the recovery routine, the routine on the right hand side of a "causes" statement. However, if the first handler also returned **false** requiring the recovery action, the recovery action caused by the later handler would have been activated with the 'wrong' values. Restrictions to control the activation of subprograms are discussed in the section on subprograms. Therefore, events of the same type get queued on their handlers, with different handlers being executed in parallel.

For illustrative purposes, let p(e),q(f),r(e) **causes** s(e,f) be a causal statement. Let events of type e with values v1 and v2 and event of type f with value w1 arrive before any handler is activated. An implementation may choose to activate p(v1),r(v1) and q(w1) in parallel. However, p(v2) can be activated only after p(v1) returns and the handler p becomes **active** again. Similarly r(v2) can be executed only after r(v1) returns and the handler is active.

4.3.8 Specific Events

As many events of the same type are generated, there is a need to identify specific instances of them. Specific instances are identified by an integer representing the instance number. To facilitate the identification in programs, a function EVENT_OCCUR is provided. It takes as argument an integer and returns the value of the event number in question. If the said event has not occurred, an error value **undef** is returned. The number of event occurrences of an event type is returned by OCCUR_NO. Therefore, for all event types e, and k greater than 0, EVENT_OCCUR(e, OCCUR_NO(e) + k) is **undef**.

A function TIME_OF returns the time a specific event occurred, with respect to a specified clock. TIME_OF has three arguments, the event type, the event number (an integer) and a clock name. The value returned is the time of event occurrence and is an integer. Similarly a function VALUE_OF returns the value associated with the specified event (an event type and an occurrence number.)

We do not allow a specification to wait for a particular event. Hence it is not possible to handle only a particular event. If this were allowed, it would be possible to write event handlers which miss certain events. As events signify important milestones, all events must invoke a handler. However, language designers one can only ensure the invocation of the handler but cannot prevent the handler from doing nothing, thereby ignoring the event.

4.4 Temporal Specifications

A language to be classified as a real-time language must support the definition of timing constraints. In this section the various forms of temporal specifications allowed by the language are enumerated. As events are to be used to mark relevant milestones during the course of the execution, the predicates in all timing constraints will refer only to event types. This implies that if a general boolean condition is to be a part of the constraints, the programmer has to perform the checking explicitly and generate an event when the relevant condition is detected. This restriction might seem strange especially when our goal was to enhance expressiveness. But one of our goals was also to design a language for executable specifications. This requires one to build a interpreter (preferably an interpreter which scales time appropriately so that mapping onto a real system becomes easier) for the language. Allowing arbitrary expressions in timing constraint statements rules out an interpreter which could scale time as it would not be possible to estimate the time necessary to execute certain statements. For example, the time necessary to execute an assignment statement will depend on the number of constraints that depend on it. The allowing of general expressions also introduces questions as to how often are the expressions evaluated? This is of relevance as the expressions refer to mutable objects. Due to aliasing (due to subprogram calls) it may not always be possible to re-evaluate the expression when a mutable object that it refers to is altered. This could result in either too many unnecessary evaluations or evaluations which are delayed. Hence it was felt that it is best to disallow general expressions as a part of a timing requirement.

Allowing only events in a temporal specification is also a consequence of our conjecture about how people design prototypes. Initially, the system designer has only a general idea of what should happen and when those things should happen. Events are used to identify the key happenings in the general setting. The designer also has a idea of by when these events should occur. Temporal specifications involving these events are then written. After that the designer writes the code to actually effect the generation and handling of events. Hence any change to the way the events are generated, need not require the re-calculations of the temporal constraints. This is because the temporal specifications are independent of the control structure of the program. This is in contrast to the situation where the temporal constraints are tied to the control flow of the program. Any change to the control flow will require recomputing of the temporal specifications. This will become clear in chapter 6 when we compare ARL with the language ESTEREL.

To define the syntax of a timing requirement, a definition of terms and operators which establish a relationship between various terms is required. In ARL real-time specifications come in two flavors. The first consists of two timing predicates and a temporal operator (the binary case) while the second consists of only one timing predicate (the unary case.) The general form is

temporal_spec :: id : predicate binary_opt
binary_opt :: limit | temp_oper predicate limit
temp_oper :: **before** | **after**
limit :: **atmost** expr | **atleast** expr

The 'label' field identifies the constraint. Its significance is explained in the next paragraph when we discuss temporal violations. In the unary case there is one timing predicate (say p) and the limit field specifies either an upper bound or a lower bound and attaches a quantitative measure. For example un : p **atmost** 5, states that predicate p *must* become true within 5 units of time. In the binary case there are two timing predicates (say p and q.) The temporal ordering operator can either be **before** or **after** and 'limit' specifies either an upper bound or a lower bound and attaches a quantitative measure. For example, ex : p **before** q **atleast** 10 specifies that the predicate p must become true before predicate q and the separation should exceed 10 units of time. The structure of the temporal predicate is discussed later.

We diverge from the main goal of this section to complete the definition of the **temporal_violation** event type. The label field acts as an identification of the specification which was violated. There are two possible types of values associated with timing errors. The first deals with the point notion of time while the second deals with an interval notion of time. The possible types of violations associated with point time, are **early, late** and **wrong_ordering**. The value **wrong_ordering** is used when events occur in an order other than that specified in a constraint i.e., the second event precedes the first. The value **early** is used if the an event occurred before it was supposed to and **late** if an event occurred later than expected. Associated with these two values is also an integer value indicating how early or late the events occurred. The error values necessary to characterize violations when interval time is used are **r_overlaps, l_overlaps, during, equals, meets, guaranteed** and **possible**. **r_overlaps** and **l_overlaps** are used to characterize overlap on the right and the left hand side respectively. **during, equals** and **meets** are as defined in [4]. **guaranteed** and **possible** are necessary as the magnitude of the separation between the intervals will vary depending on the point chosen in the intervals. **guaranteed** indicates that all points in the intervals will result in a violation, while **possible** indicates the existence of such points. The precise meaning of these values is explained in section 5.6.1.

The values described above are used only when an event actually occurs. It is also possible to detect a violation without the occurrence of an actual event. This is achieved by setting timer(s) to expire. In cases when a timer expires, **temporal_violation** is generated with value *timer_expired*. This is explained in more detail in the chapter on semantics.

The type of the value associated with events of type **temporal_violation** is TEMPORAL_LABELS × { **early, late, wrong_ordering, timer_expired, r_overlaps, l_overlaps, during, equals, meets, guaranteed, possible** } × EVENT-_TYPES × INTEGER × [(EVENT_TYPES × INTEGER) ∪ { ⊥ })]. TEMPORAL-

_LABELS is the set of labels associated with a timing specification while EVENT_TYPES is the set of event types defined in the system. The element $\perp$ indicates undefined when an event supposed to have occurred did not. In this case the second field (in the value associated with the temporal violation) has to be *timer_expired.*

We resume our discussion of temporal statements. As mentioned in the literature survey standard temporal logic operators by themselves are not quite appropriate for use in real-time computation. They do not have a strong sense of duration, nor do they deal with multiple clocks. An added problem is that operators like **eventually** while useful in specification of liveness and verification, are not ideal for real-time specification where timeliness is more important. As discussed in chapter 2, specifications involving interval logic operators like **at, starts** [4] are impossible to implement in a distributed environment. Hence, these operators also are absent in our language. The temporal operators defined by the language are **after, atleast, atmost, before,** and **wrt.** The only ordering operators are **before** and **after.** The operators **atleast** and **atmost** are used to specify the limit field, while **wrt** is used to specify the clock with whose respect time is to be measured.

To define the predicates in a timing requirement a relation called **occur** is introduced. It takes an event type, an integer and a value as its arguments The integer is to be interpreted as an occurrence number of the event type while the value field is to be interpreted as the value associated with the specific event.

The meaning of the occur relation, when used in a unary form is different from when used in a binary relation. The unary form of the occur relation specifies the type of event, its occurrence number and the value associated with the occurrence. For example, **occur**(e,i,v) by itself specifies that the ith occurrence of events of type e is *required* to have value v. When used in a binary relation, it is equivalent to stating 'if the ith occurrence of event type e has value v.' The exact meaning of the unary and binary case is discussed in chapter 5.

It is also possible to specify the clock with respect to which the time is to be measured. The specification of the clock uses the **wrt** operator. The example used above can be extended to **occur**(e,i,v) **wrt** clk, where clk is defined to be a clock. The meaning of the example extended using the **wrt** operator is that, if the ith occurrence of an event of type e has value v, then time of occurrence is measured with respect to clock clk. As will be seen in section 4.7.2, the programmer *can* specify the mapping of modules to clocks. As one knows the identity of clock associated with the module, remote clocks can be identified. The reading of a remote clock is to be interpreted as acquiring information from a remote site.

The justification for the described definition of temporal predicates is as follows. The occurrence of the ith event with value v is an indication that a significant state in a thread in the current execution has been reached. At this point one might wish to check the status at a remote site. Ideally, the status is represented by the 'state' of the site. However if the site is required to send state information, the magnitude of the message to be sent could be prohibitively large. Also, in a concurrent environment one can never be sure of a state at the time when a specification is written. It is more likely that one in a set of states is acceptable. So the subprogram requesting state information must have knowledge of all possible states. This in our opinion is also unacceptable. Thus, time at a different clock is used as an approximate measure of the status of the site which is using the clock.

It is clear that a rigid definition of the **occur** construct is not in keeping with our goals of a language for rapid prototyping. One must either design an entire class of relations similar to **occur** or must enhance the expressive power of **occur**. We have adopted the second option as we believe that fewer the constructs a language has, the easier it is to master it. Having mastered the syntax and semantics of a particular construct, it would be relatively easy to understand its variations. The language increases the power of the **occur** relation by permitting the user to 'omit' the occurrence and/or the value field in a specification. The interpretation in such cases is as if an entire class of requirements were defined. It is as if the undefined field were universally quantified. The omission of occurrence number is represented as a '*' or a '**' with '*' different from '**', while the omission of the value field is denoted by '\$' and '\$\$' with '\$' different from '\$\$'. This notation is similar to the one used to represent types in polymorphic languages such as Miranda [88]. For example **occur**(e,*,\$) **wrt** cl **before occur**(f,**,\$\$) **wrt** c2 **atmost** n is a temporal specification with all the permissible fields universally quantified. The general form of a predicate is

predicate :: **occur** (id , oc_option , val_option) time_option
oc_option :: * | ** | expr
val_option :: \$ | \$\$ | expr
time_option :: | **wrt** clock_spec_opt

In the next section, we explain all the options along with an informal meaning in detail. The precise meaning of these options is discussed in the section 5.6.

4.4.1 Temporal Predicates

Having explained the basic structure of the temporal predicates, all the permissible variations are enumerated below. As mentioned earlier, the power of the **occur** construct is increased by allowing one to 'omit' the occurrence/value field in a specification. Depending on the unspecified field, there are sixteen major categories. What follows is an enumeration along with a discussion of the intuitive meaning of each permissible variation of the **occur** relation. To do so we assume that e and f are event types, i,j are integers, and v and w are values associated with event types e and f respectively. In order to simplify the explanation, only the **before** operator is used. Also the clock(s) with respect to which time is measured is not specified, nor do we attach a numerical value to the separation required between the two events specified in the predicates. The meaning of the statements when all operators, multiple clocks and absolute values are involved is a straightforward extension to the single clock meaning. The binary use of the **occur** relation is explained at first, as it is in this case that a large number of variations are possible.

Case I: We begin by describing the case where the occurrence and the value fields are completely specified. Consider $\boxed{\text{occur(e,i,v) }\textbf{before}\text{ occur(f,j,w)}}$ as a template for this case. If the ith occurrence of event type e has value v and if the jth occurrence of event type f has value w then the e event in question should occur before the f event. If the ith occurrence does not have value v or if the jth occurrence does not have value w the specification has no effect.

Case II: In this case the second value field is left unspecified. Let $\boxed{\text{occur(e,i,v) }\textbf{before}\text{ occur(f,j,\$)}}$ be such a temporal specification. If the ith occurrence of event e has value v then it must precede the jth occurrence of event f irrespective of the value associated with the event. Notice the implicit universal quantification over the value field.

Case III: Let $\boxed{\text{occur(e,i,v) }\textbf{before}\text{ occur(f,*,\$)}}$ be a temporal specification. If the ith occurrence of event e has value v then it must precede *any* occurrence of event f. This implies that all occurrences of events of type f have to occur after the ith occurrence of event type e having value v.

Case IV: This case explains the statement where both the value fields are undefined. This case will illustrate the power of allowing implicit universal quantification. There are two possible sub cases to consider.

Let $\boxed{\text{occur(e,i,\$) }\textbf{before}\text{ occur(f,*,\$\$)}}$ be the first sub case. This statement requires that *all* occurrences of events of type f happen after the ith occurrence of event e. This requirement is independent of the value of the event in question and hence the universal quantification over *, $ and $$. The second subcase is the temporal specification of $\boxed{\text{occur(e,i,\$) }\textbf{before}\text{ occur(f,*,\$)}}$. This constraint has to be satisfied only for events of type f which have the same value as the ith event of type e. Events of type f which do not have the required value are not governed by this specification. This specification is syntactically acceptable only if the domain

of values for event types e and f have a non empty intersection.
Case V: Here all the four field are left unspecified and there are four sub-cases. Let $\boxed{\text{occur(e,*,\$) }\textbf{before}\text{ occur(f,**,\$\$)}}$ be the first sub-case. This statement is best explained by considering its equivalent in first order logic. Using the universal quantification explanation, it translates to ∀i,j,x,y occur(e,i,x) **before** occur(f,j,y). The meaning of this is: pick any (i) occurrence of event e and pick any (j) event of type f. If the specification is obeyed, then the type e event occurs before type f event. The values associated with the events are irrelevant.

The second sub-case is $\boxed{\text{occur(e,*,\$) }\textbf{before}\text{ occur(f,** \$)}}$. The meaning of this case is : pick an event of type e and an event of type f such that their event values are identical. For the specification to be satisfied, the e event should have occurred before the f event.

The third sub case of $\boxed{\text{occur(e,*,\$) }\textbf{before}\text{ occur(f,*,\$\$)}}$ translates to ∀i,x,y occur(e,i,x) **before** occur(f,i,y), i.e. the ith event of type e has to occur before the ith event of type f. The values associated do not matter.

The final sub case is $\boxed{\text{occur(e,*,\$) }\textbf{before}\text{ occur(e,*,\$)}}$. It's meaning is similar to that of sub-case three with an additional requirement that the values of the event should also be identical. Here again the types must have overlapping values for the statement to be syntactically valid.
Case VI: Consider $\boxed{\text{occur(e,i,v) }\textbf{before}\text{ occur(f,*,w)}}$ This specification requires that if the ith occurrence of event type e has value v, then all occurrences of events of type f having value w must occur after the e event.
Case VII: Let $\boxed{\text{occur(e,i,\$) }\textbf{before}\text{ occur(f,*,w)}}$ be a typical element of this case. Its meaning is: the ith occurrence of event e, should occur before all events of type f with value w.
Case VIII: We now discuss the case where both the occurrence number fields are left unspecified. There are two sub cases to consider. The specification
$\boxed{\text{occur(e,*,\$) }\textbf{before}\text{ occur(f,**,w)}}$, requires any occurrence of type e to precede any occurrence of type f with value w, while the timing statement
$\boxed{\text{occur(e,*,\$) }\textbf{before}\text{ occur(f,*,w)}}$ requires the above constraint only on events with identical occurrence numbers.
Case IX: Consider the specification $\boxed{\text{occur(e,i,\$) }\textbf{before}\text{ occur(f,j,w)}}$. This specification is operational only if the value associated with the f event is w. It requires the ith event of type e to precede the jth event of type f.
Case X: The specification $\boxed{\text{occur(e,i,\$) }\textbf{before}\text{ occur(f,j,\$\$)}}$, requires the ith event of type e to precede the jth event of type f, while the specification
$\boxed{\text{occur(e,i,\$) }\textbf{before}\text{ occur(f,j,\$)}}$ requires the constraint to be obeyed if and only if the values are identical. As before, there should be an overlap of type values

associated with event types e and f for the specification be meaningful.
Case XI The meaning of $\boxed{\text{occur(e,*,v) \textbf{before} occur(f,j,w)}}$ is the dual of case VI.
Case XII The meaning of $\boxed{\text{occur(e,*,v) \textbf{before} occur(f,j,\$)}}$ is similar to case VII.
Case XIII: Here again, there are two sub-cases to consider. As the first sub-case consider the following specification $\boxed{\text{occur(e,*,v) \textbf{before} occur(f,**,w)}}$. When expanded in logic, the statement reads ∀i,j occur(e,i,v) **before** occur(f,j,w). On picking any two events of type e and f which have values v and w respectively, the event of type e should occur before the event of type f. The second sub case of $\boxed{\text{occur(e,*,v) \textbf{before} occur(f,*,w)}}$ expands into ∀i occur(e,i,v) **before** occur(f,i,w). Pick any event of type e and the corresponding event of type f. If the e event has value v and the f event has value w, the e event occurs before the f event.
Case XIV: The semantics of the temporal requirements
$\boxed{\text{occur(e,*,v) \textbf{before} occur(f,**,\$)}}$ and $\boxed{\text{occur(e,*,v) \textbf{before} occur(f,*,\$)}}$ is similar to case VIII.
Case XV: The statement $\boxed{\text{occur(e,*,\$) \textbf{before} occur(f,j,w)}}$ has the dual meaning of case III.
Case XVI: The meaning of $\boxed{\text{occur(e,*,\$) \textbf{before} occur(f,j,\$\$)}}$ and
$\boxed{\text{occur(e,*,\$) \textbf{before} occur(f,j,\$)}}$ is the dual of case IV.

Having considered all the types of binary specifications, the reader might wonder if anything is really *required to happen.* All the above mentioned types of temporal specifications are active only if the relevant events have the specified occurrence number or value. These constraints do not require the value to be associated with the event. In other words, one might consider the above requirements as safety properties. A null program would satisfy all of the temporal conditions. It is necessary to be able to specify liveness properties or statements which require something to happen. The unary version of the **occur** relation is used for this purpose. The unary relation has a form similar to the binary relations. It has a label field identifying the specification, the **occur** predicate, an optional bound using a clock, a limit and an integer value. For example, $\boxed{\text{l: occur(e,*,v) \textbf{wrt} cl \textbf{atmost} n}}$ requires that all occurrences of event type e necessarily have the value v. Also, all the events have to occur within n units of time as measured with respect to clock cl. There are three types of the unary specification and they are enumerated below. As in the binary case we do not specify the label, the clock and the limit. The interpretation involving these fields is a straight forward extension of the interpretations of the one without the field.

Case A: $\boxed{\text{occur(e,i,v)}}$ requires that the ith occurrence of event e should have value

v. A timer is set on the start of the program to detect this if the limit field is the **atmost** operator. Note that the **atleast** operator sets only a lower bound and does not set a timer.

Case B: $\boxed{\text{occur(e,*,v)}}$ requires all events of type e to have value v. In keeping with the universal quantification which allows null occurrences no timer is set.

Case C: $\boxed{\text{occur(e,i,\$)}}$ requires the ith event to occur but the value is irrelevant. As in case A, on the start of the program, timers for specifications of this type are set appropriately.

Case D: $\boxed{\text{occur(e,*,\$)}}$ requires all events of type e to occur within the specified time. But no timers are set as a non-occurrence of event type e satisfies the requirement trivially.

4.4.2 Extensions

The '*' notation can be extended to specify events at fixed offsets. For example,

$\boxed{\text{occur(e,*+k,\$) \textbf{before} occur(f,*,\$\$)}}$

requires the ith event of type f to occur after the (i+k)th event of type e. k in keeping with our restrictions of not allowing general expressions, is required to be a compile time constant. We do not enumerate the cases involving * and *+k as these cases are straight forward extensions of cases where both event occurrence number fields are denoted by *. Similarly, the $ notation can be extended to specify values which depend on values of the relevant event. For example, $\boxed{\text{occur(e,i,\$) \textbf{before} occur(f,j,\$ } \oplus \text{ v)}}$ relates the value of the ith event of type e and the jth event of type f using a relevant operator represented by $\oplus$ and the constant v.

By implicit definition (i.e. by the implementation) of appropriate $\oplus$, pattern matching involving a single $ or a $$ in a temporal specification could be used. For example, if r is a record with two fields and if event type e takes values in r, occur(e,*,(obj_1,$)) is a valid predicate. A combination of the above such as $\boxed{\text{occur(e,*,\$) \textbf{before} occur(e,*+k,\$ } \oplus \text{ v)}}$ is also permitted. This requires extension to the grammar rule governing predicates to the following.

```
oc_option :: star_opt | dstar_opt | expr
star_opt :: * | * + expr
```

dstar_opt :: ** | ** + expr
val_option :: dollar_opt | ddollar_opt | expr
dollar_opt :: $ | $ operator expr
ddollar_opt :: $$ | $$ operator expr

In the BNF grammar described above, the exact nature of 'operator' is not specified. It can be either a pre-defined operator (like plus, minus, etc), pattern matching (as shown in the example) or a user defined subprogram. The exact BNF grammar for this can be found in appendix A.

4.5 Subprograms

The concept of subprograms is central to all programming languages. The expressiveness of a language is largely dependent on the ease of subprogram definition. That is, the techniques available to define subprograms determine the degree of ease of reading/writing programs written in the language. Care must be taken in providing tools to be used in declaring and using subprograms. In keeping with our goal of amalgamating the various programming paradigms, these tools must also help one categorize subprograms into *functions*, *observers* and *procedures*. As ARL is aimed at programmers with either a procedural or a functional background, two types of constructs for subprogram definition are provided.

The user can define all three types of subprograms as if programming in a language like Ada. In this technique, the subprogram keyword (**function**, **procedure**, **observer**) is followed by the subprogram identifier and the parameter specification followed by the body. The exact rule for a body is not given. It is the union of the rules for statements in Ada and Miranda used in the proper context. Towards the definition of the BNF rule for subprograms, let *newline* represent the lexical element carriage return. The actual BNF rule for the subprogram definition is

subprogram_decl :: subprogram_keyword_opt id parameter_spec **is** declarations **begin** body **end**
subprogram_keyword_opt :: | **function** | **efunction** | **observer** | **procedure** | **subprogram**
body :: [statement terminator]
terminator :: ; | *newline*

'subprogram_keyword_opt' permits the specification of the effect class as an indication of what is required by the programmer. If it is not specified or if the generic term **subprogram** is used the implementation will assign an effect class to it. Otherwise, the compiler will perform an effect class check and issue a warning if the specified effect class is stricter than the deduced effect class. Functions, by default, are executed in a lazy fashion. If the programmer wishes an implementation to execute a function in an eager fashion, the function should be declared as an **efunction**. The following example defines a procedure

```
procedure increment (x : in out integer) is
begin
   x := x + 10;
end;
```

The user, if familiar with languages like Miranda, might wish specify the subprogram in an equational style. Note that only the style is equational. We do not require any of the properties required of an equational programming language to be valid. The BNF-like grammar for the definition of subprograms in the equational style is

```
subprogram_decl :: subprogram_keyword_opt identifier eqn_parameter_spec
                   = body local_declaration
local_declaration :: where [ declaration ]
```

The increment example described above can also be defined in an equational style as "increment x = x := x + 10". In the equational style, pattern matching can also be used to specify the parameters to the routine. Recall the definition of type Tree. It had two constructors Nilt (for a null tree) and Node (for a node in the tree). An inorder traversal of such a tree can be specified as follows.

```
inorder Nilt = [ ]
inorder (Node x left right) = (inorder left) ++ [x] ++ (inorder right)
```

The equational technique can be used to define *functions* which are polymorphic in nature. For example if tree is polymorphic, the function inorder is a polymorphic inorder tree walking function. ARL allows only functions to be polymorphic as we do not wish to deal with the consequences of side effects in presence of polymorphism.

To retain most of the advantages of functional programming, functions should be first class objects. It should be possible to pass functions as arguments and return functions as values. To simplify matters functions are first class only with respect to functions. In other words only functions can accept as parameter or return as result functions. To achieve this all functions are *Curried*, which considers functions with more than one argument as higher order functions each with a single argument. Hence functions are first class objects i.e., can be passed as arguments as well be returned as results from functions. We explain this through an example. Recall the definition of the higher order function to implement looping

```
until final trans state = state, final state
                        = until final trans (trans state), otherwise
```

The function is polymorphic in nature with type (* $\rightarrow$ boolean) $\rightarrow$ (* $\rightarrow$ *) $\rightarrow$ * $\rightarrow$ *, where * denotes any valid type. Note that the parameters final, which determines the terminating condition and trans, the transition function, are themselves functions.

Using the definition of until, one can define a function trim which removes leading blanks as follows. Let $\cdot$ represent function composition and $\sim$ the negation function. Let 'head' return the first element of a sequence and 'tail' the last. Space is a function which returns true if the character in question is a blank or a tab. Define trim as trim = until ($\sim$ $\cdot$ space $\cdot$ head) tail

In the definition of trim the parameter '$\sim$ $\cdot$ space $\cdot$ head' is mapped onto 'final'. That is, if the first element is not a space the loop can exit. If not, the transition function is tail, i.e., the rest of the sequence. Note that the definition of until had three parameters while trim specifies only the first two. This is allowed due to Currying.

The type of trim can be deduced to be [character] $\rightarrow$ [character]. Due to the type specificity of space, head and tail, trim is not a polymorphic function but a monomorphic function obtained from until.

Note that higher order functions supported in Miranda (and hence ARL) is weaker than the higher order functions in Lisp. In Lisp one can dynamically create functions at run-time which is not permitted in ARL.

4.5.1 Nested Subprograms

One of the arguments against nested subprogram definition is that it is more expensive to implement than non nested subprograms. This is because of the need for dynamic chains along with static chains [3]. In a program development environment, it is a good idea to support nested subprogram definition. This is especially true when the *overhead* introduced by nested subprogram is relatively small when compared to other items supported by then language. It enables a programmer can use hidden subprograms to perform tasks without divulging the knowledge to others.

In our endeavor to support both the imperative style and the equational style of programming, nested definition can be achieved in two ways. In the imperative style, the subprogram is declared in the declarative region of the outer subprogram. The inner declarations can hide objects declared in its scope as shall be seen from the following example. Consider a function to sort a sequence (not the most efficient) which can be written as follows:

```
   function sort(x : seq) return seq is
       function insert(x : elem; y : seq) return seq is
{ x is used as arguments in both functions to demonstrate that one definition
   might hide another }
              tmp : seq;
       begin
           if y = null then
              y := new node;
              y.val := x;
              return y
           elsif (x < y.val) then
              tmp := new node;
              tmp.val := x;
              tmp.next := y;
              return tmp;
           else
              tmp := new node;
              tmp.val := y.val;
              tmp.next := insert(x,y.next);
              return tmp;
           end if;
       end;
   begin
       if x = null then
           return x;
       else
           return insert(x.val, sort(x.next));
       end if;
   end;
```

In the equational style, the function is defined with a **where** clause. The scoping is determined by the 'off-side rule' discussed in [88]. For example, the function described above can be written as

```
sort [ ] = [ ]
sort (a:x) = insert a(sort x)
             where
             insert a [ ] = [a]
             insert a (b:x) = a:b:x, a<=b
                            = b:(insert a x), otherwise
```

4.5.2 Restrictions on Subprogram Invocation

In this section we discuss the issues related to when subprogram can be activated. If subprograms could be executed in parallel the throughput of the system could be maximized. However, care has to be taken as otherwise an execution could lead to *non-serializable* [69] computation and also obscures the semantics of various constructs. Consider the following scenario. If the rate of event generation is faster than event handling, it is possible to invoke the handlers in parallel. These handler could be procedures which could mutate the values associated with variables. If the computation is to be *serializable*, access to these mutable objects has to be controlled. Rather than add constructs to control access to these individual variables, we add constructs to restrict subprogram activation. The rationale for this similar to choosing monitors [43] over semaphores [23].

The constraints are expressed as lists of subprogram names, called *exclusive access* lists, such that only one subprogram in it can start a thread. A thread is a subprogram invocation caused either by an event occurrence, the right hand side of a causal statement or a periodic task. This condition is less stringent than allowing only one subprogram in the exclusive access list to be active at any time. For example, if A and B are in an list and A is activated by an event occurrence, A *can* call B or A can call itself. The less stringent condition is allowed as A and B are not executing in parallel. This interpretation of exclusive access allows recursive operations on shared data structures.

For example, let A, B, C and D be procedures and { {A,B}, {C,D} } be an exclusive access list. Only one of A or B can be the start of a thread. Similarly, only one of C or D can be the start of a thread. However, it is possible to execute the two threads (say started by A and by C) in parallel. In the thread started by A, calls to B can be made.

In order to simplify the semantics (as shall be seen in the next chapter), we treat the exclusive access lists as equivalence classes. For example, { {A, B}, {A, C} } in a program is considered to be { {A, B, C} } in the semantics. This is not to say that the two are *equivalent.* Our definition could serialize an execution more than the specification. In the above example, it is possible for B and C to execute in parallel, but is dissallowed by our semantics. We let the user define the exclusive access lists individually and construct the smallest equivalence classes using from them. The syntax to define an exclusive access list is governed by the following BNF-like grammar

exclusive_def :: **excl** { id [, id] }

4.5.3 Non-Determinism

The language does not have any explicit construct to express non-determinism. However, it is possible for a program in ARL to exhibit non-deterministic behavior. The possibility of non-determinism is explained via an example. Let e be an event type that has more than one handler and suppose at least two of the handlers (say p1 and p2) are in the same exclusive access list. When an event of type e occurs, which of p1 or p2 should be invoked first is not defined by the semantics. Thus the behavior of the module on the occurrence of event e is non-deterministic.

4.6 Other Features

In this section the remaining features of the language are discussed. They include variables, types of statements and units of compilation. However, we do not describe other useful features like declaration of constants, definition of attributes for types, overloading of subprograms/operators etc. [1]. The description of such features will only be a reproduction of the Ada language reference manual. Hence the definition of ARL in this thesis can be considered to be core language to which other essential features can be added. The addition of these features should not result in a loss of expressiveness. Rather, it should enhance other facets of a language not discussed here. Thus, for a complete definition of ARL, the relevant rules(such as those for expressions, operators etc.) in Ada [1] and Miranda [88] should be added to the rules developed here.

4.6.1 Variables

If procedures are to be useful, the concept of mutable state must be defined. Two principal techniques to characterize state are 1) variables 2) exchange functions. A variable is an object whose value can be changed via an assignment statement.

Exchange functions [101] could also be used to capture the notion of state. An exchange function carries out a two way point to point synchronous communication. An exchange function has one input and returns a value of the same type as the input. A call waits for a matching call and an exchange of values occurs i.e., the input value of the matching call is returned. Consider the following example of a transaction system, where a unit sends out requests and another unit accepts these requests. Assume they use a channel called req. The behavior of the system can be represented as:

send-request(r) = req[r] and receive-request = req['null'].

When send-request is invoked (with parameter r), r is output on the channel req. An active receive-request reads from the channel, returns r (the value) to its caller and sends 'null' (indicating no information) to send-request which in turn is returned to the unit invoking send-request. By disguising the state as a subprogram call, a notation consistent with an applicative language is possible.

However, in ARL, we characterize state by variables. Variables are present in many programming languages and the user will be familiar with them. Exchange functions, while notationally consistent with functional languages will require the user to re-think and re-design implementations which are well understood. Exchange functions also introduce the notion of synchronous communication. The language will not have a uniform communication paradigm as the need to support asynchronous communication has been established. Hence exchange functions are *not* supported by ARL. All variables will be typed either implicitly or explicitly as mentioned in the section on types.

4.6.2 Statements

Statements in ARL are divided into 1) Loops 2) Conditionals 3) Assignment and 4) Subprogram call. Each of these is discussed briefly below.

Loops

Standard looping constructs such as **while, for, loop** as defined by Ada are present in ARL. We do allow the **exit** statement whose execution causes the loop to terminate. These looping statements are allowed only in *procedures* and *observers.*

For the functional part of ARL, these constructs can be defined as higher order functions and are not defined as primitives. This is because the programmer needs to be encouraged to think in terms of higher order functions when programming with functions, so as to utilize the power of functional programming. However to ease the usage of these higher order functions, an implementation should define these looping functions (while, for loop) as a part of the language environment.

Conditionals

The conditionals supported are the **if** and the **case** statements. The **if** statement comes in two flavors. The first style is used when the programmer uses an equational (Miranda) style of programming and the other is the imperative (Ada) style. In the equational style, the conditional follows ',' after the equation. For example

```
quad_solve a b c = ((-b) + radix )/(2 * a), disc > 0
                   where
                   disc = b * b - 4 * a *c
                   radix = sqrt(disc)
```

The above equation is valid only if the condition (disc > 0) evaluates to true. Otherwise a run-time error occurs. The other type of **if** follows the same rules as the Ada **if then elsif else endif**.

The **case** statement can be explicitly used only in the procedural style of programming. Pattern matching allowed in the equational style is effectively a case statement. The structure of it is identical to that in Ada.

We do not reproduce the syntactic rules for these statements. The rules for ARL is the union of rules in Ada and Miranda combined in a consistent fashion.

Assignment

The power of the assignment statement can be utilized only when programming *procedures*. Functions are allowed only single assignment to the variables defined by them. Observers on the other hand can mutate any variable defined by it but cannot change the value of any other variable. The BNF grammar governing assignment statments is

```
assign_stat :: expr := expr
```

Subprogram Call

An important aspect of a subprogram call is the evaluation of parameters. As functions are to be lazily evaluated, their parameters are not evaluated unless necessary. Calls to observers and procedures evaluate the parameters before the body of the subprogram called starts execution.

Procedures could have in, out and in-out parameters. Observers and functions have only in parameters. Subprograms which return values, defined in an equational style, have an implicit return statement viz., the right hand side of the '=' being evaluated. For subprograms defined in the usual style, a **return** statement has the same effect. The BNF grammar for a subprogram call is given below.

subprogram_call :: id (expr [, expr])
subprogram_call :: id [expr]

Comments

Two kinds of comments can be written. The first kind is a single line comment. The start of the comment is denoted by '||' and end of line terminates the comment. The second type of comment is the block comment and can extend over multiple lines. The block is enclosed in '{' and '}'.

4.6.3 Compilation Units

Compilation units permit the user to develop a program incrementally. They also allow selective modification without requiring a recompilation of the entire program. There are four basic kinds of units of compilation 1) type pool 2) event pool 3) clock pool and 4) modules. We discuss each of this in detail.

Data Type Units

A type pool is a collection of data type declarations. One type pool can use the type definitions introduced in other type pool(s) and is said to depend on them. The BNF rule governing the structure of a type pool is

type_pl_decl :: import **type_pool** id **is** [type_def] **end** type_pool_opt
type_pool_opt :: | id
import :: [**with** id [, id]]

Event Type Units

An event pool is a collection of event types. An event pool can refer to type pools to identify the type of the data field. The type of the data field can also be left undefined, in which case it should be possible to infer a type for it. The BNF rule for an event pool is

event_pl_decl :: import **event_pool** id **is** [event_type_decl] **end** event_pool_opt
event_pool_opt :: | id
import :: [**with** id [, id]]

Clock Units

A clock pool is a collection of clock definitions. As a clock definition does not depend on types, event types or other clocks, they are stand-alone units. The BNF rule for a clock pool is as follows.

clock_pl_decl :: **clock_pool** id **is** [clock_def] **end** clk_pool_opt
clk_pool_opt :: | id

Modules

A module will usually consist of the bulk of the program. It would contain local definitions, subprogram definitions, event handlers, temporal specifications and periodic tasks. It also has an initialization part which is executed soon after the clock associated with the module starts ticking.

Note the difference between the Ada definition and ARL definition. In Ada, modules can share objects and subprograms. This however is disallowed in ARL. Hence there is no need to make a distinction between the specification and the body of a module. One can interpret the lack of sharing between modules as all modules having a null specification part. The following BNF rules determine the structure of a module.

module :: import module_spec module_body init_opt
init_opt :: endm_opt | **init** statements endm_opt
endm_opt :: **end** | **end** id
import :: [**with** id [, id]]
module_spec :: **module** id **is**
module_body :: distribution_map [declarations] [subprograms |
clock_synchronization | causal_stat |
temporal_specs | periodic_tasks | event_restrict]

4.6.3.1 Parametrized Modules

We define a parallel program as a program where a number of processing elements are programmed to execute similar code. As each processing element is executing a similar code, it is unreasonable to expect a developer to program each of the processing elements individually. To support parallel programming a concept similar to a module type is introduced. A set of modules having similar behavior can be defined by parameterizing a module. For example, **module** array_process [index : 1 .. 100] ... **end** defines 100 modules having a similar structure. The module can be defined to use 'index' to exhibit different behavior. Multi-parameter modules can also be defined and are considered similar to array definitions. The mapping of this to the available hardware is not defined by language. An implementation may choose a mapping it determines to be 'ideal.' The BNF grammar for a parametrized module involves a change only to 'module_spec' and is as follows

module_spec :: **module** param_spec_opt **is**
param_spec_opt :: id | id mod_param
mod_param :: [mod_seq]
mod_seq :: id : expr .. expr [, mod_seq]

Importation

The definition of importation determines what elements declared in one compilation unit are visible to other compilation unit(s). The syntax to effect this was presented in the grammar for the compilation units. In this paragraph, we discuss it in detail.

Type definitions, except for abstract data types, and event type definitions are essentially declarative in nature, i.e., the user cannot define subprograms or periodic tasks in these units. However, the user can define initialization code for them a la Ada. The implementation will define implicit operators for actions such as allocation, selection in a record etc. Abstract data types, have functions associated with them and have an executable part. Thus replication of types involves replication of the associated functions. Clock definitions could contain clock synchronization while modules contain the bulk of the executable part of the program namely, subprograms, temporal specifications, event handlers etc. The **with** clause, as in Ada, establishes connections between the items mentioned above. Diagram 4.1 illustrates the permissible connections.

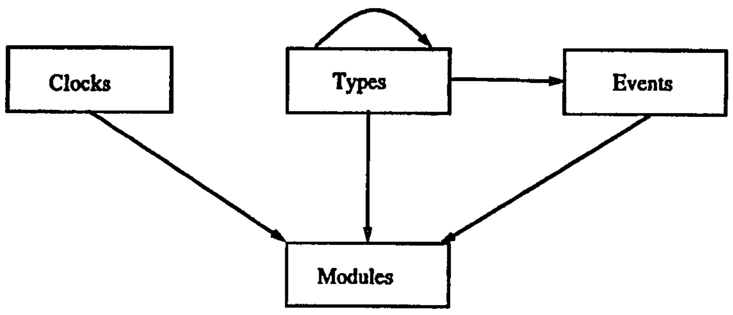


Figure 4.1. Importation Diagram

As shown in the diagram, modules can import clock, data type and event type definitions, but cannot refer to objects in another module. Therefore the modules in ARL are more restricted than packages in Ada. Our decision not to allow access to objects in other modules is discussed in section 4.7.3.

Events and types can refer to types declared in a different compilation unit. The only restriction is that complete type definition should not be circular. Incomplete type definitions as in Ada can refer to one other. Clocks cannot refer to any other item.

The **with** relation should induce a partial order on the compilation units and any total ordering of this is a valid compilation order.

In the next section some of the issues related to program termination and deadlock are discussed. Following this, we suggest a unit of distribution.

4.7 Other Issues

Issues related to termination, deadlock, divergence, units of distribution and sharing across modules are discussed in this section.

4.7.1 Termination

If we assume that a real-time program is required to maintain a continuous relationship with its environment, ideally it should never terminate. However, it should be possible to write programs which are required to terminate. In ARL, due to the declarative nature of handling events, a program terminates only when *all* of the following conditions are true.

- There are no generated events that have to be handled
- There is no active statement and as a corollary all the subprograms are idle.
- All timers that have been set (for all the unary temporal specifications and certain binary specifications) have expired. Otherwise, a timer expiration could trigger an event (**temporal_violation**) requiring a handler to be invoked.
- The communication medium is empty so that all generated events have been handled. Otherwise, it is possible for an event in the communication medium to activate a subprogram.

Related to termination is deadlock. Two modules in a system are said to be deadlocked if the following situation is reached *sometime* during a computation. Any subprogram in a module say m1, can be activated only by generating an event in another module say m2 and any subprogram in m2 is activated only by generating an event is module m1. This definition can be extended to the n-module case as follows. An n-module system is deadlocked if there is a sequence i_1, i_2, $i_3 \ldots i_n$ such that for k in 1 .. (n-1), module m_{i_k} is waiting for an event from module $m_{i_{k+1}}$ and module m_{i_n} is waiting for an event from module m_{i_1}. A system is said to be deadlocked if all its component modules are deadlocked. It is obvious that the termination condition is identical to the deadlock condition. Hence our semantics cannot differentiate between deadlock and normal termination. The interpretation of the event generation and handling has to be modified if one has to differentiate deadlock and termination. This aspect is not considered in this thesis.

However, the language does distinguish between divergence and deadlock. Recall that divergence is non-termination. The fact that the language cannot distinguish between the two is obvious as deadlock is identical to termination while divergence is the opposite of termination. Note that it is possible to differentiate between internal divergence and external divergence. External divergence occurs when the program does not terminate but continues to generate events, while internal divergence occurs when a program does not terminate but does not generate any events either. But it is not always possible to detect divergence though it would be useful to detect internal divergence and generate an event to the effect.

4.7.2 Units of Distribution

In a language designed for distributed execution, one of the first questions asked is what can we distribute or what is the unit of distribution. In certain languages, even if the distributable unit is not stated explicitly, there are some obvious candidates. However the same cannot be said for all languages. For example in CSP, processes represent units of distribution. However, in languages like Ada, if the unit of distribution and the unit of compilation are identical, the process of building a distribution is greatly simplified [95]. This, however, is not obvious from the language definition, as if one follows ones intuition, one might be tempted to use tasks — which are the units of concurrency — as the unit of distribution. Therefore, it is essential that the language defines the unit of distribution.

However, the choice of the distributable unit could depend on the nature of the underlying architecture. For example, the unit appropriate for a tightly coupled system may not be appropriate for a loosely coupled system. This has been observed in Ada, where a task is an appropriate unit of distribution in a tightly coupled system. However, tasks are far from ideal units in a loosely coupled system. However, the definition of the unit of distribution by itself is not sufficient to effect distributed execution. To execute a given program in a distributed fashion, the mapping of it to the given hardware must also be specified.

Hence we divide distributed execution into two major components. The first is the representation of distribution, while the second is the configuration or mapping a distributed program onto a given architecture. We believe that the user must be permitted to specify distribution explicitly in a program. This requires the unit(s) of distribution to be defined by the language. However the configuration issue is too implementation dependent to be solved at the language level. We discuss each of these issues in detail below.

Specification of Distribution

The concept of *virtual node* as defined in [92] is the basis of our definition of the unit of distribution. In [92] a virtual node is defined as a system consisting of multi-processor system sharing memory i.e., a tightly coupled system. A distributed system is defined to consist of a set of virtual nodes which do not necessarily share memory.

Based on this, we define modules as the unit of distribution and the basis of a virtual node. This is consistent with the definition of a distributed system as modules do not share objects (see section 4.7.3), and a module could have parallel threads of execution. A inter-module object access is in the from of events thus differentiating remote access form local access. We describe how a virtual node based on multiple clock definitions can be specified in ARL. Towards a syntax to specify distribution in a given program, let M be a module and C be a clock. The command '**for** M **use** C', associates the clock C with M. This is to interpreted as: a mapping of logical clock C to a physical clock PC, results in the mapping of module M to the processor(s) associated with PC. Define a virtual node as the set of modules which have the same associated clock.

The above syntax can be generalized so that one can specify the mapping of separate clocks to each module in a parametrized definition. For example, '**for** M[index] **use** C[index]' associates clock[index] with M[index], where M and C are parametrized modules and clocks respectively. The BNF grammar for the specifying the mapping of clocks to modules is as follows.

```
distribution_map :: for module_map use clock_map
module_map :: id | id [ id ]
clock_map :: id | id [ id ]
```

We emphasize that the user is not required to to specify the distribution. In such cases the implementation is free to choose any mapping. If a mapping is not specified for any of the modules, it is to be assumed that the programmer specified a concurrent system as opposed to a distributed system.

Configuration

Configuration or how to actually distribute a given program introduces more issues like 1) When does one specify the binding of program fragments to a processor? 2) Can the binding be done dynamically (for fault tolerance, load balancing etc.)? 3) What's the cost of such operations? 4) What are the relative speeds of the various processors?

As it is not possible to give an answer to the above questions, we refrain from defining how to execute a given distributed program in the language. Hence we also do not permit the specification of a mapping from syntactic terms to physical sites, specifically the mapping of logical clocks in the program to physical clocks. This gives the implementor the freedom to decide what actions are appropriate during the configuration phase.

In summary, ARL defines a module as the unit of distribution but it does not specify how to configure a given program. In our opinion, this is not a serious limitation as the principal concern in ARL was the ability to express various constructs. However, ARL varies from other concurrent languages in that it is possible to distinguish a concurrent program from a distributed program using the multiple clocks.

4.7.3 Sharing across Modules

It has been argued that knowledge of which object is remote is essential to design good algorithms [95]. We examine if any construct in the language language provides remoteness information.

In the current definition of the language, modules do not have access to any item defined in another module. As a consequence, a modules cannot access subprograms, or variables defined in another module. This forces events as the only form of communication between modules. The use of events will usually take considerably more time than a local procedure call. This feature can be used as information about potential remoteness. If modules are able to share variables or subprograms, one would require annotations to indicate possible remoteness. These annotations add extra syntax to the language and also has an affect on the representation of distribution. The effect of not sharing objects across modules on prototyping requires further study and is not addressed in this thesis. We emphasize that modules can 'share' subprograms defined along with abstract data types.

4.8 Conclusion

In summary, ARL is based on a paradigm derived from both functional and imperative languages. It supports type inferencing, polymorphism, lazy evaluation etc. Being a rapid prototyping language it supports the creation of dynamic objects and recursive subprogram calls. ARL also supports the definition of discrete clock(s) to be used for timing. Multiple clocks is also the focus of this thesis in denoting distribution. The user defined clocks are incremented independent of the control structure of the other elements in the system. The programmer can specify how often these clocks are to be synchronized with respect to each other. ARL supports the asynchronous communication by the way of events. Specifications involving events are used to activate new threads of control. New threads of control are essentially subprogram calls made when an event is generated. The semantics of the specification is general enough to be able to specify fault tolerance. ARL permits the definition of periodic tasks and new threads are activated when they are activated. Constructs based on the **occur** relation are to be used to specify temporal specifications. The **occur** relation can be universally quantified over certain fields thus increasing the expressiveness of the language. Though the language does not have constructs for explicit parallelism or non-determinism an ARL program can exhibit parallelism and non-determinism.

In the next chapter, we discuss the semantics of the language. Following that examples of programs in ARL and its comparison with other languages is presented.

CHAPTER 5

OPERATIONAL SEMANTICS

The first step towards building a computation model based on a programming language is to define its semantics. The two principal semantic styles are *denotational* and *operational*. The reason for choosing operational semantics over denotational semantics was presented in chapter 3. We recall some of those arguments. The underlying assumption is that most people understand the meaning of programs by visualizing behavior. If the semantics is to explain behavior of programs written in the language, an operational semantics is necessary. An operational or a behavioral view of the language captures meaning by formalizing the idea of behavior and explaining the effect of executing each syntactic entity. This is however not to conclude that other semantics styles like denotational semantics are unimportant. It is just that our current focus is on behavioral models. Further research is necessary to describe the denotational semantics along with proof rules etc. for the language to be theoretically complete. In this thesis, we develop an operational semantics based on dynamic algebras for the language.

This chapter is organized as follows. First we justify our choice of dynamic algebras over other operational styles and present an overview of the structures in a dynamic algebra. This is followed by describing semantics categories called **prototypes** which correspond to syntactic terms. The components of the initial structure or the state in which a given program starts is described. Certain assumptions about the environment in which an ARL program is expected to be executed in is described. The final section on the transition rules is composed of a discussion of time and clock synchronization, event generation and handling, the checking of the various temporal requirements and the execution of subprograms. Note that we do not discuss the semantics of types and expressions as it is only a question of re-writing the usual semantics in dynamic algebra form.

Why dynamic algebras? The following are the main reasons for choosing dynamic algebras.

- From a mathematical point of view, static properties of a system can be captured by an algebra. For systems with dynamic properties, a dynamic algebra also called an *evolving structure* is a natural choice.

- Algebraic features like abstract data types and polymorphism are difficult to describe in a purely operational setting. Dynamic algebras allows one to describe these features using their natural algebraic definition.

- In real-time systems, resources limitations play an important role. The semantic style must provide techniques to specify such limitations. In the dynamic algebra technique describing resource restrictions in actual computations is easy as they can be expressed in terms of first order formulae.

- A usual criticism of operational semantics is that, if defined completely, it is biased towards an implementation. The semantics prescribes the data structures to be used in an implementation which is not acceptable. Dynamic algebras overcome this problem by defining the relevant data structures as abstract data types. Implementation of these data types can be considered to be parameters to the semantics. This will become clearer as we describe the model.

The dynamic algebra approach has been used to describe a sequential language (Modula) [65] and a concurrent language (Occam) [37]. It is our endeavor to extend it to describe distributed real-time languages. An introduction to dynamic algebra is presented, following which the semantics for the language is developed in detail.

5.1 Overview of Distributed Real-Time Structures

An evolving structure or a dynamic algebra consists of a sequence of structures — representing states — indexed by time. Each state is composed of a finite, many sorted first order structure. This structure consists of a set of universes with an associated name also called a set of sorts and a set of functions whose domains and co-domains are universes constructed by recursively applying union and cross product to the original set of universes. A set of transition rules determine the evolution of structures into other structures.

All programs, at the start of their execution have an initial state/configuration. This initial configuration usually depends on the program. The initial configuration is modified to reach subsequent configurations by the transition rules. For the sake of explanation, assume that we are given a program P. A configuration contains of 1) A set of Σ algebras (defined below). The configurations of an ARL machine have a Σ algebra for each data-type in the program P disambiguated appropriately. 2)

A set of statements that can be executed in the configuration 3) A set of temporal constraints (specified in P) that are to be satisfied/verified.

The formal definition of Σ algebra is as follows. Define a *signature* as a set of *sorts* S and a set of operators Σ and a function from Σ to $(S^* \times S)$. Associated with the signature can be a set of axioms which the operators satisfy. In programming language parlance, S is the set of types, Σ the set of operators on the type. The function of the signature assigns types to the operators. The axioms represent the properties satisfied by the operators. Formally, a Σ algebra $\mathcal{A}$, is an S-indexed family of sets A, with an operation $\sigma_{\mathcal{A}}$: $A_{s1} \times \ldots \times A_{sn} \rightarrow A_{s'}$ for each σ in Σ of type $s1 \times \ldots \times sn \rightarrow s'$ satisfying the defined axioms. The notion of Σ algebras is a well studied topic and we refer the reader to [32, 100] for further details.

Returning to the definition of configuration, one might wish to include behavioral specifications relating the multiple clocks defined in P if the network characteristics are known. This is discussed in section 5.3.3. One can also add resource restrictions like memory size, cpu speeds, the largest/smallest possible integer etc to the initial configuration. These restrictions, would be specified by first order formulae. For example, MEM_AVAIL = 100,000000 could be used to indicate that the memory available is 100M units. These restrictions are not necessarily static and could change from one state to the other. For example, the invocation of a subprogram could reduce MEM_AVAIL by the size of the subprogram's activation record.

As we cannot envisage all the resource restrictions, an implementation will necessarily define more restrictions than defined here. In such a case, the transition rules we develop here have to be extended to model the new resource availability formulae.

Certain items which are not directly relevant to the core of the semantics need not specified by the language designers. For example, the data structures used by the abstract machine need not be specified completely. In fact, if the semantics is to be unbiased towards an implementation, the data structures necessary *should not* be specified in detail but should be axiomatized. Such items are considered to be parameters to the semantics. In other words, different behaviors of these parameters could result in different program behaviors.

The meaning of a program, given an initial configuration and instances of the parameters, is the set of all possible evolutions from the initial configuration which satisfy the transition rules. This will become clearer when the semantics are discussed in their entirety.

5.2 Protostructures

If the number of transition rules is finite, the usefulness of the semantics will increase greatly. For example, finiteness could be used in the development of a semantic driven compiler, would simplify any theory of equivalences etc. It is clear, that we cannot consider individual transition rules for all possible syntactic terms as this would result in infinite rules. In order to make the set of rules finite, the transition rules should be treated as a relation on a set of syntactic terms.

It is obvious that similar constructs have similar transition rules. That is, the essential actions necessary to execute similar constructs will be similar. For example, the transition rules explaining different event generation will almost be identical except for the name of the event the value of the event, the time of event instantiation etc. But the basic meaning of event generation, which is transmitting the information to all sites that can handle it, does not change.

One could classify the infinite set of syntactic elements into a finite number of classes. These classes are called *protostructures* or *prototypes* [37]. Formally, a *protostructure* is a class of evolving structures corresponding to some syntactic category in the language. Different structures of a *prototype* will have a common set of transition rules and integrity constraints or similar functionality. But they are likely to have different sets of constants, resource restrictions, universes and functions. Functions which permit the differentiation between instances of the prototypes are also defined in the semantics.

For example, the prototype **add** can be used to characterize the addition operation. The transition rules shall describe how to effect the addition. However, the prototype **expression** which could describe the set of all legal expressions has no transition rules as there is no necessary common behavior between the elements of this class. Different elements of the **add** prototype could have functions like *arg1* and *arg2* indicating the values of input variables. These functions will have different behaviors depending on the binding of arguments of the instance of **add**.

5.2.1 Prototypes for ARL

Here we enumerate the ARL *prototypes* and identify the universes and functions associated with elements of the *prototypes.* The principal prototypes are **modules, expressions, statements, subprograms, events, clocks, causation, temporal_specifications, clk_sync periodic_tasks**. The **statements** prototype can be further subdivided into **if, case, while, for, assignment, generate** and **reply.**

The **temporal_specifications** prototype is composed of prototypes for the sixteen main cases and the numerous other sub cases, while the **expressions** prototype will consist of prototypes for all the standard operations like +, -, function call etc. Let PROTO be the set of all ARL prototypes. Also construct a function TERM_TYPE which when given a syntactic term returns the prototype to which the syntactic term belongs to. For example, TERM_TYPE(**auto** id := id + 1) returns **clocks**.

5.2.2 Functions on Prototypes

Having defined a finite number of prototypes, certain identifying functions which differentiate one instance from another are defined. Associated with all instances of every prototype is a function MY_SELF which returns the identity or name associated with the prototype. The function MY_SELF has no argument and varies from instance to instance. For example, in the transition rules dealing with clocks, MY_SELF return the name of the clock used in the syntactic term. We describe these identifying functions by declaring their type and describing their behavior. We emphasize that these identifying functions defined do *not* operate on *all* prototypes. They operate only on certain prototype(s) as shown in the definition below.

- Clocks
  - INCR : **clocks** → INTEGER
  - INITIAL : **clocks** → INTEGER
- Causation
  - CAUSER : **causation** → $\mathcal{P}$ (**subprograms**)
  - CAUSED : **causation** → **subprograms**
  - CAU_EV : **causation** → $\mathcal{P}$ (**event_types**)
  - EVENT_SCOPE : **causation** × **event_types** → $\mathcal{P}$(**subprograms**)
- Temporal Specification
  - FIRST_EV : **temporal_specification** → **event_types**
  - SECOND_EV : **temporal_specification** → **event_types**
  - FIRST_OC : **temporal_specification** → INTEGER ∪ { *, ** }
  - SECOND_OC : **temporal_specification** → INTEGER ∪ { *, ** }
  - FIRST_VAL : **temporal_specification** → **values** ∪ { $, $$ }

- SECOND_VAL : **temporal_specification** → **values** ∪ { $, $$ }
- FIRST_CLK : **temporal_specification** → **clocks**
- SECOND_CLK : **temporal_specification** → **clocks**
- LIMIT : **temporal_specification** → { UPPER, LOWER }
- LIMIT_VAL : **temporal_specification** → INTEGER
- TS_LAB : **temporal_specification** → **temporal_labels**

- Generate
  - GEN_NAME : **generate** → **event_types**
  - GEN_VALUE : **generate** → **values**
- Clock Synchronization
  - SYNCER : **clk_sync** → **clocks**
  - SYNCED : **clk_sync** → **clocks**
  - SYNCINT : **clk_sync** → INTEGERS
  - B_CLK_SYNC : **clocks** → $\mathcal{P}$(**clocks**)
- Subprograms
  - SPGM_TYPE : **subprograms** → { Procedures, Observers, Functions, Efunctions }
  - SPGM_PARAM : **subprograms** → $\text{INTEGER}_{\textbf{(expr)}}$
- Periodic tasks
  - PT_INTERVAL : **periodic_tasks** → INTEGERS
  - PT_NAME : **periodic_tasks** → **subprogram**
  - PT_CLK : **periodic_tasks** → **clocks**
  - PT_PARAM : **periodic_tasks** → $\text{INTEGER}_{\textbf{(expr)}}$

5.2.2.1 Description

Type definition by itself does not define a function. The exact nature of the above declared functions is now elaborated. Towards this description, consider the following.

- Let **auto** clk := clk + n **init** k be an instance of **clocks.**
- Let p(e),q(f) **causes** s(e,f) be an instance of **causation** and be labeled *cau* for the purposes of future reference.
- Let **generate**(e,v) be an element of the **generate** prototype. Label it *gen* for future reference
- Let **auto** (clk1 >> interval) (clk2 := clk1) to be an element of the **clk_sync** prototype. Let *sync* stand for the instance specified above.
- Let **auto**(c1,c2,c3: b_interval) be an example of multi-clock synchronization. Let *bsync* represent the instance.
- For an element in the **temporal_specification** prototype consider $\boxed{\text{label : } \textbf{occur}\text{(e, oc1, v1) } \textbf{wrt} \text{ c1 } \textit{op} \ \textbf{occur}\text{(f, oc2, v2) } \textbf{wrt} \text{ c2 } \textit{lim} \text{ t\_error}}$, where oc1, oc2 and t_error are elements of INTEGERS ∪ { *, ** }, v1 and v2 elements of VALUES ∪ { \$, \$\$ }, *op* ∈ { **before, after** } and *lim* element of { **atmost, atleast** }. Let label be an identifier for the entire specification. Recall that *, **, \$ and \$\$ denote universal quantification in temporal specifications.
- Let p(e1,e2) be an element of the **subprogram** prototype. Call it proc for purposes of identification.
- Let **auto** (clk >> rate) (p_task a1, a2, ... an) be a periodic task labeled ptask for further reference.

The behavior of the functions is enumerated below.

- INCR(clk) = n
- INITIAL(clk) = k
- CAUSER(*cau*) = { p, q }
- CAUSED(*cau*) = s
- CAU_EV(*cau*) = { e, f }
- EVENT_SCOPE(*cau*,e) = { p }
- GEN_NAME(*gen*) = e
- GEN_VALUE(*gen*) = v

- SYNCER(*sync*) = clk1
- SYNCED(*sync*) = clk2
- SYNCINT(*sync*) = interval
- B_CLK_SYNC(c1) = { c2, c3 }
- SYNCINT(*bsync*) = b_interval
- TS_LAB(label) = label
- FIRST_EV(label) = e
- SECOND_EV(label) = f
- FIRST_OC = oc1
- SECOND_OC = oc2
- FIRST_VAL = v1
- SECOND_VAL = v2
- FIRST_CLK = c1
- SECOND_CLK = c2
- LIMIT(label) = UPPER if *lim* = **atmost** else LOWER
- LIMIT_VAL(label) = t_error.
- SPGM_PARAM(proc) = { (e1,1), (e2,2) }
- PT_INTERVAL(ptask) = rate
- PT_NAME(ptask) = p_task
- PT_CLK(ptask) = clk
- PT_PARAM(ptask) = { (a1,1),(a2,2) ... (an,n)}

5.3 Initial Configuration

The meaning of of a program execution depends on the state in which the computation starts. The start state has to be well defined to understand the meaning of a program. In dynamic algebra terminology, the start state is referred to as the initial structure. In the following paragraphs we describe in detail the components of the initial structure.

As in VDL [99], we assume that a program is compiled into an appropriate form on which the transition rules operate. To capture the compilation phase of a program, we assume the existence of a parse tree which is used by the transition rules. Also assume that the associated universe for this parse tree is $\mathcal{U}$. Define $\mathcal{U}$ to have a set of nodes (NODES) which are instances of elements of PROTO and static functions TERM_TYPE, ROOT, PARENT. We also define a dynamic function called NEXT. TERM_TYPE, as defined before returns the name of the prototype to which a syntactic term belongs. The other functions return various executable statements. The function ROOT of a term points to the first executable statement in the term, while PARENT of a term points to the first executable statement block enclosing the term. NEXT points to the next executable statement. For the sake of explanation consider, the following code fragment.

```
L0 j := true
L1 while (e) loop
L2        x := 10
L3        y := f(e,j)
end loop
L4 ⋮
```

Let L0, L1, L2 and L3 represent pointers to the appropriate instances of the relevant prototypes in the parse tree. ROOT of the sub-tree is L0. PARENT of L2 or L3 is L1 and NEXT of L2 is L3. The NEXT of L1 will either be L2 or L4 depending on whether e evaluated to **true** or **false** respectively.

To formalize the flow of control from one statement to the other, dynamic constants in a universe called MODE is introduced. The values in the universe MODE are **working** and **dormant**. In the initial configuration, the root of each of the modules in the program is in mode **working** and all other nodes have their mode as **dormant**.

All transition rules that are discussed in this chapter have the structure described below. Let '$[\![$ execute body $]\!]$ ' represent the transition rule(s) for the appropriate body.

```
if MYSELF.mode = working then
    ⟦ execute body ⟧
    NEXT.mode := working
    MYSELF.mode := dormant
end if
```

The check for the **working** state and the flow of control are *not* written explicitly when the transition rules are developed. Transition rules that describe the 'run-time' system activities are explicitly activated by other transition rules and do not have the "working" check. For example, the liveness and reliability condition of the communication medium, invokes the RECEIVE_MESSAGE transition rule. The RECEIVE_MESSAGE transition rule does not check the **working** condition. This will become clear when the transition rules are defined.

Certain components of the initial structure are composed of sub-structures. The emphasis in this section is on the static structure reflected by a program. The items we discuss are modules, structure to represent time, multiple clocks, clock synchronization, temporal specifications and events. The dynamics of the program is captured by the transition rules and is not discussed in this section. Before describing the machinery on which the parse tree is based, we define a number of finite sets constructed from the syntactical definition of the program.

5.3.1 Sets

A program in ARL consists of a collection of named modules. It is only natural that we define a set MODULES representing the syntactic modules. Each element of a parametrized module is considered to be a separate and different element of MODULES. Subprograms are defined in modules, due to which we construct a set SUBPROGRAMS containing the names of *all* subprograms in the system disambiguated in an appropriate fashion. This set can be partitioned into equivalence classes with each class containing subprograms defined in a particular module. It can also be partitioned on the basis of its effect class being either a efunction, function, observer or a procedure. Let E be the equivalence class obtained from the set of all exclusive access lists in the program.

The set EVENT_TYPES contains all the event types defined in the program, while the set CLOCKS contains the names of all clocks declared. The set VALUES represents the union of all possible value domains. Assume the definition of sets like INTEGER, REAL, CHARACTER and BOOLEAN which are subsets of VALUES. Also defined are sets EXPR, the set of all expressions, EVHAN_SPECS the set of all event handler specifications, TEMPORAL_LABELS the set of of all labels associated with the timing requirements. The set PTSK_LABELS defines the set of labels associated with periodic tasks in the program.

The semantics uses other sets, which are defined when necessary so as to justify their need. We also define functions which are constructible from the syntactic structure of a given program. These functions are based on the above defined sets. A point worth noting is that these functions are static, i.e., do not change with time. Later on we shall introduce what are called dynamic functions whose specifications undergo change with the passing of time. In other words, the specification of the dynamic functions are affected by transition rules and they capture state information.

A function MY_MODULE: NODES $\rightarrow$ MODULES returns the module in which the argument (which corresponds to a syntactic term) is present. A function VISIBLE is defined as MODULES $\times$ EVENTS $\rightarrow$ BOOLEAN. It returns true if the event type parameter is visible in the module for input. The event type is either local to the module or it has been 'withed' and not restricted to **output**. A MODULES indexed function SUBPR_SCOPE: SUBPROGRAMS $\rightarrow \mathcal{P}$(EVENT_TYPES) is defined. It returns the set of event types that could activate the given subprogram. This value is set of all event types that occur as parameters to the subprogram used in the causal statements in the module. SUBPR_SCOPE is the universal (with respect to the module) scoping function and not restricted to a specific causal statement. In a later section, when we discuss the semantics of the causal statement, a function which returns the scope of an event for a given causal statement is defined.

5.3.2 Structure of a Module

As an ARL program consists of a set of modules, which encapsulate various other constructs, it is natural to start describing the initial structure of a module. That is, the structure of the module defines the environment in which the program operates.

The primary executable unit in a module is a subprogram, due to which the execution of subprograms determines the dynamics of a module. As certain subprograms can be executed in parallel, a module at any given time, can have multiple threads (subprograms) of execution. Thus a parallel automaton is necessary to represent the behavior of a module. This parallel machine can be represented by a set of automata (we consider them to be sub-automata.) However, parallelism in a program can be restricted by the definition of exclusive access lists. The semantics has to respect the mutual exclusion defined by these lists.

Towards a formal definition of this, let M be a module and E_M be the set of equivalence classes representing the set of exclusive access lists defined in M. Let $S = \{ p_1, p_2, \ldots\ p_n \}$ be the set of all procedures and observers that are in any exclusive access list in M. All procedures that do not occur in any of the explicit lists form singleton sets in E_M. For E_M to be meaningful, all elements of E_M are pairwise disjoint. In other words, no p_i can be in more than one set. Therefore each element of E_M is an equivalent class, with up to one element in an equivalence class being active at any given time. Let k be the cardinality of E_M. Therefore the automaton representing the module has *atleast* k sub-automata. Let $A' = \{ a_1, a_2, \ldots\ a_k \}$ be the set of automata representing S.

It would appear that if one had an automaton per subprogram, the semantics would be simplified. This however is not the case. The reason being that when an automaton, corresponding to a subprogram, say p, is started, all the other automata corresponding to subprograms in p's E_M-equivalent class have to be disabled. This is to respect the mutual exclusion specification i.e., atmost one of these can be active at any given time. Transition rules for a procedure call would then involve disabling the automata and at the end of the call, *all* the automata in the equivalence class have to be re-enabled. Concurrent threads activated by procedures in an exclusive access list have to be queued. Note that the queues associated with the automata are not required to be FIFO. The exact handling of the queues, i.e., the selection of which thread to activate, is left as a scheduling decision and is a parameter to the semantics. So, no advantage is lost by considering a single automaton per equivalence class. The single automaton can be thought of having a case statement which branches out to execute the appropriate sub-automaton.

A' by itself is not sufficient as S does not contain all the subprograms in the module. Observers not mentioned in E_M and all functions can have multiple threads active at a time and be executed in parallel. In order to model finite resources, an implementation of the language usually bounds the number of threads that can be active at any given time. The resource bounds can be represented by an integer which indicates the maximum number of threads that can be simultaneously active. This integer could vary from implementation to implementation depending on the memory available, speed of the CPU(s) etc. and is to be treated as a parameter for the semantics. Let the number of the threads allowed in a module be THREADS. Let CURR_ACTIVE denote the number of threads that are active. An obvious restriction is CURR_ACTIVE $\leq$ THREADS.

As multiple threads of functions and observers are permissible, it is possible that THREADS number of automata for functions/observers are necessary. To keep track of the number of current threads started by a function, a dynamic constant

LOCAL_THREAD is maintained with each instance of the **subprogram** prototype. Clearly, LOCAL_THREAD $\leq$ THREADS for all subprograms and for **procedures** LOCAL_THREAD is always ≤ 1.

Therefore the automata which simulates a module can be defined as A = A′ ∪ {THREADS copies of the each of the automata executing functions and observers not represented in A′ }. In the initial structure, all the automata are in a *ready* state and CURR_ACTIVE is 0. All the functions and observers have their LOCAL_THREAD set to THREADS (could be set to 0 without any major change.) The *ready* state is explained in detail when the transition rules for subprograms are discussed.

5.3.3 Time

The technique we adopted to model time is a generalization of the technique described in [54]. In [54] discrete time using a single clock is represented by a structure $\mathcal{C} = (\omega, \leq, +)$. In the above structure, ω is the set of natural numbers, $\leq$ the natural (and obvious) well ordering on ω and $+$ the successor operator. We, having chosen to represent distributed systems using multiple clocks, cannot use this simple structure. Our model cannot ignore the differences between single and multiple clock systems as it is the sole feature characterizing distribution. To define a suitable structure which represents multiple clocks accurately, we consider the operations affecting the value of clocks 1) incrementing the value of current time and 2) clock synchronization. The meaning of incrementing can easily be represented by a successor operator (denoted by $+$). However clock synchronization that invalidates the simple structure.

In the single clock case, the clock value can be assumed to increase monotonically. But since we are dealing with multiple clocks and have to model clock synchronization it is inappropriate for clocks to satisfy the monotonic increasing condition. This is because clock synchronization could 'reduce' the value of current time. This raises the possibility that at two different points in time with respect to a reference clock, a clock could read identical values. Therefore the clocks are piecewise monotonic. Also the possible values of a clock does not form a set but a multi-set. A clock in a distributed clock environment is modeled by a structure $\mathcal{C} = (\omega_c, <_c, +)$. ω_c is a multiset with preferably finite multiplicity. $<_c$ a well ordering on ω_c. An obvious choice is be ordering the function on its domain. '$+$' the increment operator. However the '$+$' operator has to reflect clock synchronization. Every clock in the system has a structure similar to the one described above.

The various clocks can either be related abstractly by defining functions between the various structures or operationally by developing transition rules to explain the required relationships. The set of functions relating the various structures, is described in the next section. For an operational description of a clock, we create a unique process that simulates it. As a particular clock can be assigned to a particular module it is also possible to have the process as part of the automata executing the module. The operational view of multiple clocks in the form of transition rules are described in a later section.

5.3.4 Multiple Clocks: Functional Relation

An important operation that needs to be supported is 'knowing' the time at a remote site. Define a function called REMOTE_TIME_FUNCTION to achieve this. It takes three arguments viz., the clock name, time with respect to that clock and the clock name whose time we wish to determine. As one is never certain of the time at any remote site, a range of values representing the set of possible times (with respect to the clock named in the first parameter of the function) is essential.

In a program, one could either deal with interval time or with point time. Though point time is easier to use, there is a loss of information when converting from interval time to point time. In this section, we define REMOTE_TIME_FUNCTION to deal with point time and not a range. The discussion on ranges is presented in section 5.6.1.

To arrive at a point notion of time, we define REMOTE_TIME_FUNCTION to return an 'average' value within a range of integer values by using a function called AVERAGE. The function AVERAGE is to be considered as an input parameter to the semantics as we as language designers do not have sufficient information to define it completely. Therefore, the semantics is not biased towards any particular implementation.

To achieve a consistent relation between the various clock definitions we introduce a meta clock (MC). The meta clock could be distinct from any other clock defined in the system. It could also be equal to a clock defined by the program. The first case will arise when one uses universal clocks like UTC to relate the various clocks, while the second situation will arise when a clock in the system is designated as the principal clock. To cater to both these cases, the initial value of MC is defined to be 0 and its increment to be 1. Therefore, MC is ω (the set of naturals) well ordered the natural way.

Functions which capture the meaning of time as determined by multiple clocks are defined. They are also parameters to the semantics in that we do not (and

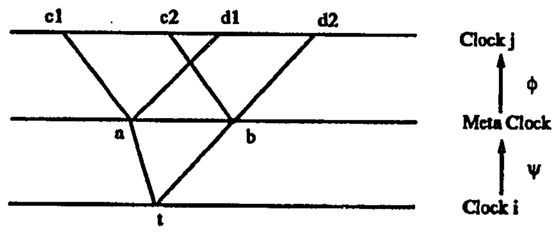


Figure 5.1. Remote Time Function

cannot) specify their precise behavior. However, we discuss the various constraints on them. Define a CLOCKS indexed family of functions ($i \in$ CLOCKS) ϕ_i : MC $\rightarrow$ ($\omega_i \times \omega_i$) satisfying the following properties

- $\forall$ n $\in$ MC if ϕ_i(n) = [x,y] then x $\leq$ y

- if m $\leq$ n then $\forall$ j $\in$ {1,2}: ϕ_i(m)(j) $<_i$ ϕ_i(n)(j)

In the description above, [x,y] is to be interpreted as defining the range of values of the clock i when MC reads n i.e. the range of values within which the value of clock i lies. The first property requires the range to 'good' while the second property requires both the first and second fields in the range of clock values to be monotonic under the ordering $<_i$.

Similar to the ϕ's, we define a CLOCKS indexed family of functions (i $\in$ CLOCKS). These functions are like inverses for the ϕ's, i.e., map a clock value onto a range in the domain of the meta-clock MC. ψ_i : ω_i $\rightarrow$ MC $\times$ MC. If ψ_i(x) = [a,b] then a = min {y : x in ϕ_i(y) } and b = max {y : x in ϕ_i(y) }.

As an example of the above consider ϕ to be $\{ <0,[0,0]>, <1,[1,2]>, <2,[2,3]>, <3,[2,4]> <4,[3,4]>, <5,[3,6]>, <6,[4,5]> \}$. Then ψ(3) = [2,5]

The behavior of the REMOTE_TIME_FUNCTION can now be specified as shown below. Its pictorial representation is shown in figure 5.1.

REMOTE_TIME_FUNCTION i t j = AVERAGE(min(c1,c2),max(d1,d2))
where
ψ_i(t) = [a,b], ϕ_j(a) = [c1,d1], and ϕ_j(b) = [c2,d2]

5.3.5 Clock Synchronization

Clearly the behavior of ϕ's and the ψ's has to depend on the synchronization constructs used in the program. That is, they cannot be completely arbitrary functions but need to satisfy certain constraints. These constraints depend on the program in question. We illustrate the constraints via an example. Recall that synchronization of clocks C1 and C2 is syntactically expressed as **auto** (C1 >> n)(C2 := C1). An descriptive meaning of the specification is: Every n ticks with respect to clock C1, the current value of the clock C1 is copied into clock C2. This copying need not be (and usually is not) instantaneous and hence there is an error associated with the time indicated by clock C2. That is to say, when C2 is assigned the value from C1, C1 would have changed its value. Let ψ_1 and ϕ_1 be associated with clock C1 and ψ_2 and ϕ_2 be associated with clock C2. A plausible characterization of the delay in clock synchronization is described below.

- $\forall$ k >0, $\exists$ ϵ_{min} such that ψ_1(k*n) $\subseteq$ $\psi_{2,1}$(k*n+ϵ_{min}) and

- $\forall$ k >0, $\exists$ ϵ_{max} such that ψ_1(k*n) $\subseteq$ $\psi_{2,1}$(k*n+ϵ_{max}).

In the above notation $\psi_{i,j}$ denotes the constraint on ψ_i when synchronized with clock j. See figure 5.2 for a pictorial representation of the constraint.

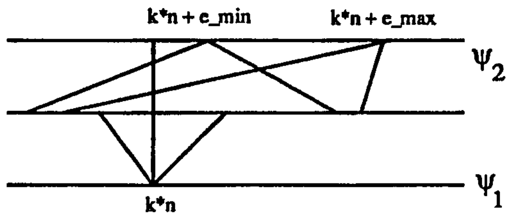


Figure 5.2. Clock Synchronization

The above requirement states that whenever there is a synchronization to be performed (signified by k), there is a defined interval within which the actual synchronization is achieved. The bounds on this interval is denoted by ϵ_{min} and ϵ_{max}. These parameters will depend on when the synchronization is being effected. That is, the parameters are a function of n and k and other factors such as network characteristics. The above constraints, define a relation between the various ψ's. In other words the ψ's are not independent.

The ψ's and the ϕ's have to satisfy the constraints imposed by *all* the clock synchronization specified. Therefore ψ_i will be the outer envelope of all the individual $\psi_{i,j}$'s. Formally, let $\psi_{i,j}$(n) = [a_j,b_j] for all j. Then ψ_i(n) = [min{ a_j }, max{ b_j }]

Notice that the ϕ's, the ψ's and the f's are not completely defined and hence are parameters to the semantics and are a part of the initial configuration. Also note that we have not prescribed a clock synchronization algorithm. We have only shown how to characterize the effects of clock synchronization. The chosen algorithm along with the network characteristic will determine the ϕ's and the ψ's precisely.

5.3.6 Temporal Ordering Consistency

Another important aspect of ARL is the presence of temporal constraints. For each module, a data structure called the *temporal_verifier* is constructed. It stores all the temporal constraints that have to be satisfied. It would be extremely inappropriate for the semantics to dictate the precise construction of the table. However the table has to satisfy certain properties which can used by the transition rules. These axioms define the table as an abstract data type.

Each entry consists of the name of the specification (i.e. an element of TEMPORAL_LABELS), type of first event its occurrence number and value, the clock name with which to measure time for the first event, the type of the second event and its occurrence number and value and the clock name associated with it, the relation between the events and the numerical value associated with the relation and whether it is the upper bound (denoted by **atmost**) or lower bound (denoted by **atleast**).

Whenever a temporal specification is encountered (at compile time) in the program, it is entered into the *temporal_verifier*. This table, fully constructed with all the specifications, is a part of the initial configuration. Whenever an event occurs the *temporal_verifier* is notified and it acquires the relevant information and performs appropriate checks depending on the operators involved in the actual specification. If the check fails it generates an event to signify *temporal_violation*. The exact nature of the information acquisition and checks performed are governed by transition rules which are described in a later section. For example, the specification

label: **occur**(e,*,v) **wrt** cl **before occur**(f,i,$) **wrt** c2 **atmost** n

results in the entry <label, e, $\perp$, v, cl, f, i, $\perp$, c2, before, upper_bound, n> being added to the data structure. If an event of type e with value v occurs, the transition rules to verifying the above requirement are activated.

5.3.7 Timer and Interrupts

To make a real-time system reliable and rugged, ideally, one should be able to signal timing violations 'as soon as possible.' When a real-time language is implemented, timers are set to expire after a specified interval. The expiration of the timer indicates that time has elapsed without the occurrence of the relevant event. A realistic semantic model must be sensitive to timers. Note that this per se does not specify any lower/upper bound on when a violation will be detected.

However, when timers are coupled with temporal specification verifiers, it is possible to generate 'potentially incorrect' (or not totally correct) warnings. Consider the scenario where an event has actually occurred but the handler was interrupted before entering the information into a data structure as a timer expired. The handler for the timer expiration detects that the event has not occurred and signals a temporal violation.

As we discuss later, our semantics maintains a table of events and their time of occurrence. To avoid race conditions similar to the one described above, an event is said to have occurred only when it is entered into the table and all timers waiting on it reset. This is an acceptable definition as event occurrences are atomic.

To simulate a timer with operations of setting and reseting the structure associated with the clocks has to be enriched. We define a timer to be a queue which is sorted on time of *expected* events. That is, the first element in the queue is the event which is *expected* to occur at the earliest, the second is the one is the next etc.

Define a timer $\mathcal{T}$ to a *ordered sequence* of type EVENT_TYPES $\times$ INTEGER $\times$ INTEGER $\times$ TEMPORAL_LABELS, with operations SET and RESET. Let *queue* be an instance of $\mathcal{T}$. The *ordering* of *queue* is defined as $\forall$ i $<$ j : Let *queue*[i] = (e_i, o_i, t_i, l_i) and *queue*[j] = (e_j, o_j, t_j, l_j) then $t_i \leq t_j$. Two procedures SET and RESET with parameter type EVENT_TYPES $\times$ INTEGER $\times$ INTEGER $\times$ TEMPORAL_LABELS are defined. Assume they operate on instances of $\mathcal{T}$.

Assume for the moment that there is only one timer called *queue* and that SET and RESET operate on it. SET(e,i,t,l) results in *queue* being altered such that $\exists$j : *queue*[j] = (e,i,t,l). In other words sets adds the entry to the queue. RESET(e,i,t,l) results in *queue* being modified such that $\forall$j : *queue*[j] $\neq$ (e,i,t,l). As a shorthand let *top* be defined as *queue*[1].

A timer, to function correctly, requires co-operation from the clock. The timer is said to expire when the 'current_time' shown by the clock is greater than or equal to the time field of *top*. When a timer expires it is as if " **generate (temporal_violation(**l, **timer_expired**,e,i))" were executed. In other words, the expiration of a timer is an indication of an occurrence of a timing error.

In the initial configuration, timers are set for all unary timing requirements defined in the program. Timers will be set in certain binary cases as shall be seen in section 5.6.

5.3.8 Events

In ARL, events are the only form of communication between the various modules. Recall, that events belonged to a group of elements called the event type. To keep track of the occurrence number of each event, every instance of the event prototype has a module indexed dynamic constant OCCUR_NO which indicates the occurrence number of the next event. It is essential to have the constant indexed by the set of modules, as different modules might handle/generate different events. It is also necessary to keep track of *all* the events that have occurred. To do so define for each module(denoted by M), a EVENT_TYPES indexed function called M.EVENT_OCCUR: (POSITIVE $\rightarrow$ ((VALUES $\times$ CLOCKS $\times$ POSITIVE) $\cup$ { $\perp$ })) where POSITIVE to be the set of non-negative integers. Given a module M and an event type e, M.EVENT_OCCUR(e) is a function which when presented a positive integer, to be interpreted as an occurrence number, returns the value and time associated with it if the said event has occurred. If the specified event has not occurred $\perp$ indicating undefined is returned.

For example, let *e :: arbiirary_type* be such a declaration. M.EVENT_OCCUR(e) is a function from POSITIVE to (*arbitrary_type* $\times$ CLOCKS $\times$ (POSITIVE $\cup$ { $\perp$ })). One can use *arbitrary_type* instead of VALUES as the programs are assumed to be type correct. In the case when a type error is detected, the program terminates with an appropriate error message. Note that type errors can arise because of heterogeneous types.

A natural restriction on EVENT_OCCUR which forces the occurrence number to be assigned in order is:

$\forall$ M $\in$ MODULES and e $\in$ EVENT_TYPES :
 if M.EVENT_OCCUR(e)(n)=$\perp$ then
 M.EVENT_OCCUR(e)(n+1)=$\perp$

For example, let M.EVENT_OCCUR(e)(20) be $\perp$ and M.EVENT_OCCUR(e)(19) be not $\perp$. The above constraint forces the occurrence number of the next event to be less than or equal to 20, e.g., 21 is disallowed. The less than (i.e. reusing an occurrence number) is avoided by choosing the minimum k such that M.EVENT_OCCUR(e)(k) is $\perp$.

At the start of the program, no event has occurred in any module due to which M.EVENT_OCCUR(e) returns undefined (or $\perp$) for all modules, event types and integers i.e.,

$\forall$ M $\in$ MODULES, i>0, e $\in$ EVENT_TYPES: M.EVENT_OCCUR(e)(i) = $\perp$.

Summary

This concludes the discussion of the initial structure. To summarize, it consists of a set of automata which represent the subprograms. If possible the behavior of the various clocks and their synchronization is captured by the ϕ's and ψ's with appropriate constraints. Data structures to store timing requirements and the events that occurred are also defined. In the next section certain assumptions about the environment in which the programs operate are discussed.

5.4 Prelude to Transition Rules

Having described the initial structure of the various elements of the language, we are almost ready to discuss the transition rules. However, one cannot discuss the meaning of programs in vacuum. The behavior of a program depends on its interaction with the environment. This is especially true in the case of real-time systems which try to maintain some harmony with the environment. Different behaviors will be elicited by different environments. For a program to execute 'correctly', the behavior of the environment must be well defined. These assumptions about the environment describes what the language expects the nature of the external world to be.

5.4.1 Environment

The conditions discussed below, specify the behavior of the environment in which *all* programs execute. While we use temporal logic to specify the behavior of the environment, it can be translated into first order logic to be consistent with our definition of dynamic algebra. The environment constitutes of the behavior of the program defined clocks and the message system. Note that we cannot characterize all types of environment here and what follows is to be considered as a example.

A clock defined in the program has to be incremented at a regular frequency by all implementations. Otherwise the intuitive definition of a clock is lost. Thus we require that the clocks actually tick, which is specified in temporal logic as

Axiom 5.1 $\forall$ C $\in$ CLOCKS $\forall$ n $\in$ ω: $\Diamond$ (C.current_time $\geq$ n)

In the above axiom, 'current_time' is a dynamic constant which is modified by the clock to indicate the passage of time. The axiom requires that a clock can always attain a time which is greater than any constant there by forcing it to tick.

As discussed in chapter 4, there are two possible interpretations to the clock definition. One was the notion of frequency and the other was the notion of clock resolution. The axiom(s) governing the behavior of clocks depends on the assumptions made. Under all circumstances, the axioms defined in section 4.2.2 hold.

If the notion of frequency is chosen, and if the mapping of the logical clock onto a hardware clock is assumed, all clock increments must not be slower than any other task that can be performed. This is to give a realistic picture of hardware clocks. Since the basic task is computation of expressions, the following condition that clock incrementing is no slower than expression evaluation, also needs to hold.

Axiom 5.2 $\forall$ C $\in$ CLOCKS $\forall$ e $\in$ EXPR: $\Box$((C $\leadsto$ C+INCR(C)) **not after** uneval(e) $\leadsto$ eval(e)).

$\leadsto$ is a causal relation derived from the transition rules by iteration (Kleene *). a $\leadsto$ b can also be interpreted as a is true before b and b eventually becomes true. The above formula requires all clocks C to attain its next value (i.e., be incremented by INCR(C)) no later than the time when expression e is completely evaluated. The axiom for the resolution case was discussed in section 4.2.2.

As the basic form of communication (between modules) in the language is asynchronous, the semantic model also needs to assume only an asynchronous paradigm. We assume a simple send-receive paradigm for communication. Note that the language does not define the topology of interconnection between the various modules. As we are considering dynamic structure semantics for the language, the topology of the network can be used as a parameter. The behavioral restrictions dictated by the topology can also be specified axiomatically when specifying the behavior of the environment As our immediate goal is only to define semantics for the language, we do not wish to characterize the internal details of an unreliable communication medium. Unreliable communication can be modeled by having additional axioms to the basic set of axioms discussed below.

Towards the specification of a channel define MESSAGES to be set of messages that can be transmitted. This channel characteristic is specified by

Axiom 5.3 $\forall$ mess $\in$ MESSAGES, N $\in$ MODULES, $\forall$ M $\in$ MODULES:
N.SEND_MESSAGE(mess) $\rightarrow \Diamond$ M.RECEIVE_MESSAGE(mess).

The above specification requires that all message which are sent will eventually arrive at *all* modules. The above rule can be optimized by identifying modules which do not needing the message and not sending the message to them. The protocol used and whether the transmission required error detection and multiple sending is irrelevant at this stage as communication medium characterization is not the primary goal of this research. However, we would also like the communication medium not to generate any spurious (or duplicate) messages. This can be specified by:

Axiom 5.4 $\forall$M$\in$MODULES: M.RECEIVE_MESSAGE(mess) $\rightarrow$
$\exists$M'$\in$MODULES: M'.SEND_MESSAGE(mess) **before**
M.RECEIVE_MESSAGE(mess)

As mentioned earlier, the set of axioms defined here are by no means complete. They only demonstrate the power of dynamic algebras. Any other environmental parameters that are deemed necessary can be used in a similar fashion.

5.5 Transition Rules

The transition rules for each syntactic entity in the language can be developed now. Before doing so, we explain briefly how to interpret the rules. All transition rules have two parts to them: an *antecedent* and a *consequent*. The *antecedent* will be a boolean test and the *consequent* a set of statements to be executed. If the evaluation of the test results in **true** the consequent is activated and executed. The antecedent part of the transition rules will be omitted where it follows the structure described in section 5.3. The transition rules also need to satisfy a *liveness* requirement, so that the rules actually execute. We shall assume that they are a part of the semantics and shall not state them explicitly here. The transition rules for all the language constructs viz., time, events, temporal specification and subprograms are discussed in the following sections.

5.5.1 Time

In ARL, the notion of time is defined by a program itself instead of a reference to an external notion of time. This notion of time is obtained by using *clocks*. All *clocks* defined in a program have two main fields viz. the initial value and an increment value. Semantically every clock (for example clk), which is an instance of the prototype **clocks**, has a dynamic constant called *current_time* which keeps track of the current time. It is updated periodically, at a rate consistent with the axiom governing clock increments. The speed of increment is governed by the environment axioms and the liveness conditions of transition rules. The following assignment statement characterizes clock increment.

$$current\_time := current\_time + \mathrm{INCR(clk)}.$$

5.5.1.1 Operational Relation Between Multiple Clocks

When discussing the initial structure, we assumed that clock indexed functions such as the ϕ_i and ψ_i could be defined with respect to some meta-clock. But it is not obvious that such functions could easily be defined. Those functions were defined to give a mathematical feel for a multiple clock system.

In this section, we present a purely operational semantics for relating the various clocks in the system. In fact the presentation of the purely operational definition is much simpler as there are no assumptions and there is no need to explain various function characteristics and justify why it is reasonable to assume a certain behavior. But the drawback of this scheme is that the behavior of the various clocks has to be derived from the transition rules. Also the operational description must be general enough so that it can enable the characterization of various schemes to synchronize clocks.

5.5.1.2 Transition Rules Relating Multiple Clocks

A few words about the approach we take to keep track of remote time are in order. There are two operations that need to be described to fully explain the relation between the various clocks. The first operation involves storing times which are equivalent. For example, when clock cl reads t one should know what the associated value with clock c2 is. This mapping is used by the transition rules defining the meaning of the temporal constraints. This use will become apparent when we consider the following timing requirement.

occur(e,i,v)wrt cl before occur(f,j,w) wrt c2 atmost n

As the programmer cannot specify the assignment of clocks to syntactic units, the meaning of the above statement must be defined so as to be independent of the clock it is assigned. Assume that a clock c be associated with the timing requirement. It seems inappropriate for one to wait till the ith instance of event e occurs (say at time t as measured with respect to c) to send a message to clock cl to find out the time corresponding to t. It would be better if one had a map which could be consulted as the result is obtained quickly. This is true if 'n' were small and the jth instance of f occurred before the value corresponding to t were obtained. In such a case the delay in signaling a possible violation could be unacceptable. Note that the actual value returned by either strategy does have an associate error which cannot be eliminated completely. Hence a map between various clocks is constructed by periodically sending out a message requesting time and storing a value dependent on the returned result.

The second operation in a system with multiple clock is to return the requested mapping from one clock to another. It involves a look up and if the corresponding value is not present, some interpolation (or extrapolation) is done. Towards a precise formulation of the above remarks, define two message types CLK_VAL and CLK_RESP. CLK_VAL indicates the message is requesting the clock value, while CLK_RESP is the acknowledgement to that and associated with it is the clock value (an integer). The polling for remote time is specified by

```
∀ C in (CLOCKS - { MY_SELF }): [ SEND_MESSAGE(CLK_VAL,C)
                                 sent_time[C] := MY_SELF.current_time ]
```

The above rule sends a message to all other clocks requesting their time. The time when the message was sent is noted. The frequency of the execution of this rule is also governed by the liveness axiom for this rule. The SEND_MESSAGE causes RECEIVE_MESSAGE to be executed by the receiving module. On checking that it is a request (CLK_VAL) to it (MY_SELF) the receiving clock sends a reply as follows.

```
if RECEIVE_MESSAGE(CLK_VAL, MY_SELF) then
  REPLY_MESSAGE(CLK_RESP, MY_SELF,current_time)
```

This message is received only by the original module, as the response was sent as a REPLY. The original module executes

```
if RECEIVE_MESSAGE(CLK_RESP,C,T) then
  ADD_TO_TIME_TABLE(C, ESTIMATION( current_time, sent_time[C]), T)
```

As message passing time could be significant, the time returned would have been attained somewhere between the time the message was sent and the time the message was received.

In the above rule ADD_TO_TIME_TABLE is a function which stores the entry in a data structure for retrieval. ESTIMATION is a function which calculates an 'average' value for the local time. The function ESTIMATION takes in two arguments and returns a value. The type of value returned depends on the nature of time necessary. If a single integer value represents time, the function returns an integer. Otherwise an interval is returned. This value represents an estimate of the

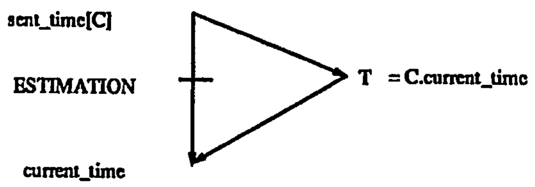


Figure 5.3. Remote Clock Values Estimation

local time. The returned value is then stored in the table. As message passing time depends on the underlying hardware/network, the semantics does not prescribe an estimation function and ESTIMATION is a parameter to the semantics. A plausible ESTIMATION function for an integer time characterization is

$$(\text{current\_time} + \text{sent\_time}[C])/2$$

with the assumption that message passing takes approximately the same time either way and the mid point is a better estimate of what the local time was when the remote clock read the returned value. This is shown in figure 5.3.

Now to define REMOTE_TIME_FUNCTION, a function which allows one to project time with respect to other clocks. Recall that REMOTE_TIME_FUNCTION takes a clock and time with respect to it as the first two arguments. The third argument it a clock whose time is to be determined.

```
REMOTE_TIME_FUNCTION C1 t C2 = V
        where X = INTERP_FROM(C1,t) and V = INTERP_TO(C2,X).
```

In the above definition, INTERP_FROM and INTERP_TO are interpolation functions. They also are to be treated as parameters to the semantics.

We present an example of these functions below. To define INTERP_FROM(C, T), let T0 be the largest value in the table with respect to clock C (used by ADD_TO_TABLE) less than or equal to T and T1 be the smallest value in the table with respect to clock C greater than or equal to T. If T1 is not defined, let T0 be the second largest and T1 to be largest values in the table. T0 and T1 represents the range within which T lies. Let the corresponding values with respect to the local clock be A0 and A1 respectively. INTERP_FROM(C,T) is defined as

```
if T=T0 then
  return A0
else
  return A0 + [ (A1 - A0)(T-T0)/(T1 - T0) ]
```

INTERP_TO(C,T) behaves similarly but maps onto clock C instead of from C. Note that the above definition of the interpolation function, a point time value characterization is assumed. For an interval characterization of remote time, the functions INTERP_FROM and INTERP_TO can be defined by interpolating the start and the end of the interval.

The rest of the behavioral rules use REMOTE_TIME_FUNCTION and are unaffected by the type of characterization. That is to say, it is independent on the way multiple clocks are defined. This concludes our discussion on how to read time from a remote clock.

The other activity involving multiple clocks is clock synchronization. Recall, that associated with the prototype for clock synchronization were functions SYNCER, SYNCED and SYNCINT. There are two transition rules for this prototype one associated with SYNCER and the other with SYNCED. The SYNCER has a dynamic constant old_sync_time which represents the the last time a clock synchronization was effected. The transition rule executed by SYNCER is

```
if [ (current_time - old_sync_time) ≥ SYNCINT ] then
    old_sync_time := current_time;
    SEND_MESSAGE(CLK_SYNC, SYNCED, old_sync_time)
```

The SYNCED clock has no dynamic constant but executes the following transition rule

```
if (RECEIVE_MESSAGE(CLK_SYNC, MY_SELF, T) then
    current_time := SYNC_FUNC(T)
```

The function SYNC_FUNC returns a time based on its argument, which accounts for network time and other errors and is a parameter to the semantics.

The transition rules described above can be extended to handle broadcast synchronization (messages from N clocks) before deciding when to synchronize and the value after the synchronization. Such schemes are discussed in [51, 80]. Recall that the syntax for such a scheme is **auto** (id [, id] : expr). These schemes can be abstracted by the following transition rules. The rules consist of two parts. One, is to broadcast its time after the synchronization interval has elapsed. This is captured by

```
if [ (current_time - old_sync_time) ≥ SYNCINT ] then
    old_sync_time := current_time;
    ∀ C ∈ B_SYNCED(MYSELF): SEND_MESS(B_CLK_SYNC, C, MY_SELF,
                                      old_sync_time)
```

In the above rule, B_CLK_SYNC, represents that broadcast semantics is used, while B_SYNCED(C) is the set of clocks which are kept in synchrony with C. The message is stored by the receiving clock(s). When 'sufficient' information has been received by the clocks (signified by a function called SUFF returning true), the time is updated. The transition rules for updating the clocks are as follows.

```
if (RECEIVE_MESS(B_CLK_SYNC, MY_SELF, C, T) then
    GOT_FROM := GOT_FROM ∪ {C}
    TIME_FROM[C] := [ T, current_time ]

if SUFF(GOT_FROM) then
    (current_time := SYNC_FUNC(TIME_FROM))
    GOT_FROM := {}
```

5.5.1.3 Discussion

The main advantage of the transition rules based characterization over the function based approach is that it is a purely operational. Hence a semantics driven compiler can be implemented directly. The implementor need not provide any functions to be used in determining temporal violations due to which the implementation need not check for validity of the functions during the course of execution.

The disadvantage is that for temporal analysis the relevant functions must be constructed. This is because it is usually easier to do an analysis using functions than using transition rules. At this point, it is not evident that functions can easily be derived from the transition rules without further assumptions.

5.5.2 Events

Events have a many fold use in the language. They can be used to specify any asynchronous behavior and communication aspects of distributed real time systems. They also form the basis for the timing requirements imposed on the system specified. This important role in the language requires one to develop the theory of events in complete detail. Recall that one actually defines event types and instantiates a number of events of that type. Recall also that an event type declaration is of the form *event_name :: type*, where *type* is a data type defining the set of values that can be associated with the value field of an instance of the event type.

To assign an occurrence number to the an event that occurs in a module, an EVENT_TYPES indexed dynamic constant called OCCUR_NO is maintained. For example, if E is an event type, E.OCCUR_NO is the integer to be used as the occurrence number for the next event of type E. To keep track of event occurrences, an MODULES × EVENT_TYPES indexed function EVENT_OCCUR was defined. For an event type and an integer namely the occurrence number, EVENT_OCCUR returned the value and time associated with the event if the event has already occurred, returning $\perp$ otherwise.

Events are generated by executing a **generate** or the **reply** command. These statements take an event type and a value of the appropriate type as arguments. The effect of **generate** is to inform *all* modules which can handle the event, that the event with the associated value has occurred, while the effect of **reply** is to send the message only to the module that generated the event that caused the activation of the current thread. When a module detects an event occurrence, all relevant subprograms in the causal statements are activated. All the relevant temporal constraints are also checked.

There is a prototype associated with the **generate** statement. Let p be an instance of the **generate** prototype. Let e stand for GEN_NAME(p) and v stand for GEN_VALUE(p). Define a message type EV_GEN to denote event generation. A message of this type has associated with it the module name of the destination, the source module name, the type of the event and the value associated with the event. When p is to ready be executed (i.e., the event is to be generated), the following transition rule is executed

```
∀ M ∈ MODULES such that VISIBLE(M,e) :
        SEND_MESSAGE(EV_GEN,M, MY_MODULE(MY_SELF),e,v)
```

The above rule, sends an event generation message to all modules which can handle the event (denoted by M) with values e and v. The sender identifies itself as the originator of the message. This is essential in the case when the responder for this event used the reply command. The transition rule for the **reply** prototype with e the GEN_NAME of the prototype and v the GEN_VALUE is

```
SEND_MESSAGE(EV_GEN,REPLY_TO,MY_MODULE(MY_SELF),e,v)
```

The parameter REPLY_TO is the name of the unit (event from a module, a causal statement or a periodic task) which caused the activation of the current thread and is set when a thread is activated. If REPLY_TO is undefined, the **reply** statement executes the rule associated with the **generate** prototype. Therefore, the transition rule for the **reply** prototype is

```
if REPLY_TO = ⊥ then
```

```
    ∀ M ∈ MODULES such that VISIBLE(M,e) :
            SEND_MESSAGE(EV_GEN,M,MY_MODULE(MY_SELF),e,v)
else
    SEND_MESSAGE(EV_GEN,REPLY_TO,MY_MODULE(MY_SELF),e,v)
```

Due to the liveness requirement of the communication medium, RECEIVE-_MESSAGE will be executed by each module to whom the message was sent. The action subsequent to the execution of RECEIVE_MESSAGE is described by

```
if RECEIVE_MESSAGE(EV_GEN,M,FR_PK,e,v) & ( M = MY_SELF) then
    EVENT_HANDLER(FR_PK,e,v);
    TEMPORAL_CHECKER(e,v)
```

The procedure EVENT_HANDLER stores the event value and time of occurrence in a table, while TEMPORAL_CHECKER performs consistency check of the temporal specifications involving the event. The semantics of EVENT_HANDLER are discussed in the next section while the rules specifying the behavior of TEMPORAL_CHECKER are discussed in section 5.6.

Before discussing the semantics of EVENT_HANDLER, the event along with other information like the time of occurrence needs to be stored in a table. The first step towards this is measuring the time and storing it in a temporary variable, i.e., *temp* := *current_time*. The actual storing is achieved by changing the dynamic function EVENT_OCCUR to a new EVENT_OCCUR called EVENT_OCCUR′ for definitional purposes. Recall that EVENT_OCCUR keeps track of the events, their values and time of occurrence. Towards the definition of EVENT_OCCUR′, define k to be E.OCCUR_NO. If occurrence numbers are assigned in order, E.OCCUR_NO has to be the minimum integer such that EVENT_OCCUR(e)(k) = ⊥. The new EVENT_OCCUR is defined as follows.

$$\text{EVENT\_OCCUR}'(e)(n) = \begin{cases} (v, clock\_assign(M), temp) & \text{if n = k} \\ \text{EVENT\_OCCUR}(e)(n) & \text{otherwise} \end{cases}$$

In essence, EVENT_OCCUR(e) is a structure which stores the information associated with *every* event occurrence. The occurrence of an event results in it being updated in the natural way. Following the modification of EVENT_OCCUR, the OCCUR_NO is incremented by 1.

5.5.2.1 Event Handler

Having stored the event, its handler(s) i.e., subprograms (identified in *causal statements*) waiting for the event to occur are to be activated. This is achieved in two steps. The event value is first queued on each of its handlers. This is to take care of the situations when the handler could be in the **passive** state for some time or was taking more time than the event generation rate. In either case it may have a backlog of events to be handled. When an event is ready to be handled, it is promoted to a queue associated with the subprogram. This is essential as it is possible that no permissible thread of control is available to execute the required invocation and the automaton corresponding to the subprogram cannot be activated.

To formally specify this, more definitions are introduced. For semantical purposes, we label uniquely each subprogram in every event handler statement. This set of labels called EVENT_LABELS, is disjoint from the labeling of timing requirements. Another set CAU_LABEL is defined to identify the CAUSED, in all handlers. The set PTSK_LABELS defines the set of labels associated with periodic tasks in the program.

Associated with every instance of the **subprogram** prototype is a dynamic constant WHO_CALLED which indicates who called the subprogram. This is necessary to provide the correct semantics for the **reply** statement. WHO_CALLED takes a value in the set EVENT_LABELS $\cup$ PTSK_LABELS $\cup$ SUBPROGRAMS $\cup$ CAU_LABEL. If WHO_CALLED is an element of SUBPROGRAMS it is an indication that the subprogram was called by some other subprogram and not via an event occurrence or a periodic task. An event invoked subprogram call sets the WHO_CALLED to the appropriate EVENT_LABEL. A periodic subprogram call has its WHO_CALLED as an appropriate element of PTSK_LABELS.

Also defined is a function EVHAN_STATE, for event handler state, with one argument from EVENT_LABELS and returns a value in the set { **active, inactive, passive** }, i.e., it returns the state of the subprogram in a given event specification. Furthermore, define a function EVHAN_ID to be a routine, which when presented a subprogram a causal statement and an event returns an element in EVENT_LABELS identifying the subprogram EVHAN_ID can be typed as **causation** $\times$ **subprograms** $\times$ **event_types** $\rightarrow$ EVENT_LABELS. Also define a function PROC: EVENT_LABELS $\rightarrow$ SUBPROGRAMS to return the subprogram associated with an event label. Note that CAU_ID is different from EVHAN_ID as CAU_ID returns an CAU_LABEL when presented a causal statement.

For example, let p(e),q(f) **causes** r(e,f) be a causal statement. Call it *spec* for the purpose of explanation. Define the set EVENT_LABELS to contain Lab_p_e and Lab_q_f, such that EVHAN_ID(*spec*,p,e) is Lab_p_e and EVHAN_ID(*spec*,q,f) is Lab_q_f. PROC(Lab_p_e) is the subprogram p and PROC(Lab_q_f) is the subprogram q.

Let $\{g_i(e_i)\}$ be a typical specification to handle events i.e., an instance of the prototype **causation**. Assume one of the e_i's is e. The following transition rules describe the behavior on the occurrence of an event of type e. For the purpose of explanation assume the specification to be labeled *spec.* Recall that EVENT_SCOPE was defined as a function from **causation** × EVENT_TYPES → $\mathcal{P}$(**subprograms**). For a given specification and event type, it returned the set of subprograms which could be activated. The transition rule governing the behavior of **causation** under an event of type e is presented below.

$\forall$ g $\in$ EVENT_SCOPE(spec,e): ADD_TO_LIST(el,(v,FR_PK))
    where el = EVHAN_ID(g,spec,e)

In order to have a complete and precise semantics one must specify the transition rules governing the behavior of ADD_TO_LIST. If one is to define transition rules for ADD_TO_LIST a specific data structure must be used. It is inappropriate for the semantics to dictate specific data structures and we refrain from providing transition rules for ADD_TO_LIST.

An additional reason for not defining the exact semantics, is not to define the order in which the events are to be handled. This flexibility provided to an implementation allows it to choose a program dependent (possibly intelligent) scheduler to generate a schedule which could reduce the number of temporal violations. The nature of scheduler should and will determine the exact nature of ADD_TO_LIST. Each module might require its own scheduling policy and hence might have its own version of list manipulating routines. For example, an implementation may assign priorities to each event and maintain a priority queue. On the other hand a simple scheduler might maintain a FIFO queue.

However, ADD_TO_LIST cannot be left completely undefined. Intuitively, ADD-_TO_LIST adds an element to an existing data structure. The first argument identifies the data structure and the second indicates a value. Axioms governing the data structure and operations have to be defined. We develop these axioms after all the related operations on the data structure in question have been defined. To make the semantics complete, the set of all ADD_TO_LIST's can be considered to be parameters for the initial configuration. In other words, the meaning of a program is indexed by the specification of ADD_TO_LIST's.

Two items are discussed in the following paragraph. The first being the execution of the subroutine on the occurrence of an event, the other being the activation of the right hand side of the **causes** relation when all the elements on the left hand side are **passive**. We present the transition rules for these in order.

Having placed the task on the event handler queue, our concern in this section is to promote it to the subprogram queue. As this should be done only one at a time, we use the state **inactive** to indicate that the handler is waiting for the subprogram to return before the next event can be handled. The transition rules regarding subprogram calls are defined in section 5.6.2 and will describe the complete behavior.

```
∀ el ∈ EVENT_LABELS such that EVHAN_STATE(el) = active:
      EVHAN_STATE(el) := inactive
      ADD_TO_LIST (Q_LAB(PROC(el)), (v,FR_PK,el))
                            where REMOVE_FROM_LIST(el) = (FR_PK,v)
```

In the above rule, Q_LAB is a function identifying the data structure for the appropriate subprogram (identified by PROC of the event label). Just as ADD_TO_LIST was not specified completely, REMOVE_FROM_LIST will also not completely specified. It will be considered as a parameter to the semantics. Its effect is to remove an element from the data structure into which it was added by ADD_TO_LIST.

The conditions that ADD_TO_LIST and REMOVE_FROM_LIST have to satisfy are given below. They are stated in terms of pre and post conditions with an additional requirement that the statement terminates. The notation for the axioms is the precondition followed by a horizontal line followed by the statement followed by another horizontal line followed by the post condition.

Define $\mathcal{L}$ to be the data structure operated upon by ADD_TO_LIST and REMOVE_FROM_LIST. Let t be a sequence and a typical instance of $\mathcal{L}$. Let $\mathcal{I} = \{ i : t[i] = y \}$ i.e., all the elements of the structure with value y. Let $\mathcal{I}'$ be the value attained by $\mathcal{I}$ after executing the relevant (ADD_TO_LIST or REMOVE_FROM_LIST) procedure. Let $| S |$ denote the cardinality of set S. The properties satisfied by the routines ADD_TO_LIST and REMOVE_FROM_LIST are

Axiom 5.5 $$\frac{\dfrac{|\mathcal{I}| = N}{\text{ADD\_TO\_LIST}(t,y)}}{|\mathcal{I}'| = N + 1}$$

Axiom 5.6 $$\frac{\dfrac{0 < |\mathcal{I}| = N \;\&\; \text{REMOVE\_FROM\_LIST}(t){=}y, \text{ where } (y = t[j])\ j \in \mathcal{I}}{\text{REMOVE\_FROM\_LIST}(t)}}{\mathcal{I}' = \mathcal{I} - \{j\}}$$

The first axiom requires that ADD_TO_LIST results in the element being added to the data-structure, while the second axiom requires that REMOVE_FROM_LIST actually deletes the element from the queue.

Before exiting each subprogram checks the corresponding WHO_CALLED. If WHO_CALLED is an element in SUBPROGRAMS or PTSK_LABELS, the subprogram terminates as usual. If it belongs to EVENT_LABELS, the called subprogram sets the state of the subroutine of the appropriate label to either **active** or **passive** depending on the value of the computation. If the subprogram activation was due to the right hand side of a causal statement (belonging to the **causation** prototype), which is indicated by the value of WHO_CALLED in CAU_LABEL, and if the value to be returned is not **true**, an event of type **disaster** is generated. The transition rules for this is discussed in section 5.6.2.

Having discussed the activation of the left hand side of the causal statement, we discuss the activation of the right hand side of the causal statement (say *spec*). The right hand side is activated when all the subprograms on the left hand side have become **passive**. The execution of the right hand side is nothing but queuing a request to the appropriate subprogram. The transition rule to activate the right hand side is as follows

```
if [ ∀ e ∈ CAU_EV(spec) & ∀ g ∈ EVENT_SCOPE(spec,e):
          EVHAN_STATE(EVHAN_ID(g,spec, e)) = passive ] then
   ADD_TO_LIST (Q_LAB(CAUSED(spec))), ( E_PARAMS(spec),
                                       CAU_LAB(spec), CAU_ID(spec))
```

The antecedent of the rule checks that *all* the handlers in the causal statement are in the **passive** state. If so, the right hand side (also a subprogram, denoted by Q_LAB(CAUSED(*spec*))) is activated by placing a request onto the queue of the appropriate subprogram. The use of CAU_LAB(*spec*) (in the WHO_CALLED field) indicates that the semantics of the **reply** statement if used is the same as **generate** statement. This is because the subprogram invocation is not due to an event occurrence but rather due to a causal statement. This concludes our discussion of event generation and handling. In the next section we discuss the semantics of temporal constraints which are constructed using event types.

5.6 Transition Rules for Temporal Requirements

A real-time language must necessarily have features to define timing requirements which are to be satisfied by the program. Such features were discussed in chapter 4. The semantics of these timing requirement statements have to be defined with care, if the work has to be the basis for a model of real time computation. We define two types of semantics for the temporal constraints as we had two kinds of definitions for time. The first is assuming a point definition of time, i.e., using the 'AVERAGE' function. The second style defines the meaning of operators using an interval meaning of time. In this section we describe in detail the transition rules associated with the various timing specifications using the point definition of time.

The timing constraint checking is a process involving a number of data structures. Once again, we shall not specify the exact nature of these data structures but define them in terms of the properties they satisfy.

As the temporal specifications involve events, we define a data structure composed of events. Recall that events belonged to a type. Each event also has a instance (or occurrence) number, a value and time of occurrence associated with it. Towards the formalization of the above, define a type EV_TAB as EVENT_TYPES $\times$ INTEGER $\times$ VALUES $\times$ CLOCKS $\times$ INTEGER. The first INTEGER field indicates the occurrence number while the second INTEGER represents the time as measured with respect to the CLOCK field. The function EVENT_OCCUR defined before returns an element of type EV_TAB.

The temporal checker needs to identify all the events which are 'relevant' to it. For example, an event of type e, with value v is relevant to a timing specification involving **occur**(e,*,v) and is not relevant to a specification involving **occur**(e,*,w). Towards the characterization of the above, define an operation GET_EVENT_ENTRY. Its parameter is an element of EVENT_TYPES $\times$ (INTEGER $\cup$ $\{\perp\}$) $\times$ (VALUES $\cup$ $\{\perp\}$) and the value returned by it are elements of EV_TAB. The $\perp$ in the above definition represents unspecified. GET_EVENT_ENTRY returns the set of events which match the input parameters. Essentially, GET_EVENT_ENTRY returns a subset of an appropriate projection of the function EVENT_OCCUR. This is stated axiomatically as follows.

Axiom 5.7 GET_EVENT_ENTRY*(f,j,val) = S where*
$\forall x,y$ $<f,j,val,x,y> \in S$ *iff* EVENT_OCCUR$_f(j) = (val,x,y)$

Axiom 5.8 GET_EVENT_ENTRY*(f,,val) = S where*
$\forall u,x,y,w$ $<f,u,val,x,y> \in S$ *iff* EVENT_OCCUR$_f(u) = (val,x,y)$

Axiom 5.9 GET_EVENT_ENTRY*(f,i,) = S where*
$\forall u,x,y$ $<f,i,u,x,y> \in S$ *iff* EVENT_OCCUR$_f(i) = (u,x,y)$

Axiom 5.10 GET_EVENT_ENTRY*(f,,) = S where*
$\forall i,w,x,y$ $<f,i,w,x,y> \in S$ *iff* EVENT_OCCUR$_f(x) = (w,x,y)$

In the first axiom, all the three parameters to GET_EVENT_ENTRY are defined. Thus the event to be returned is completely defined. GET_EVENT_ENTRY returns a null set if the event in question has not occurred. In the second case, the instance number is left unspecified. GET_EVENT_ENTRY returns all events of the specified event type that have the value 'val'. In the third case, the ith instance of the specified event type is returned, while in the last case, all events of the specified type are returned. These axioms essentially state that the function EVENT_OCCUR is a memory and GET_EVENT_ENTRY returns the set of relevant elements which have already occurred.

The temporal checker also needs to know *when* an instance of an event type occurred. Towards this, define a function TIME_OF of type EVENT_TYPES $\times$ INTEGER $\times$ CLOCKS $\rightarrow$ INTEGERS. It returns the time a specific event occurred with respect to a specified clock and is defined as follows

```
TIME_OF(e,i,c) = t
        where
              <e,i,v,c',t'> ∈ GET_EVENT_ENTRY(e,i,)
              if c=c' then t=t' else
                t = REMOTE_TIME_FUNCTION(c', t,' c)
```

Transition Rules

Having discussed the functions used by the transition rules, we describe in detail the rules for the temporal checker. There are two sets of transition rules per type of timing statement. One set of rules deals with the occurrence of the first event, while the other deals the the occurrence of the second event. Recall that for all modules M, an event occurrence results in the incrementing of e's occurrence number and storing the event value and the time of occurrence via modifying the function EVENT_OCCUR.

As there are two sets of transition rules per specification type, it is possible to detect violations at two places. Due to this it is conceivable that two violations for the same misordering of events be signaled. For example, if e is required to occur before f and if it so happens that f occurs before e, the transition rules handling the occurrence of f will detect an error as e has not yet occurred. Similarly the transition rules for event e could detect an error that f has already occurred. Such multiple detection of the same violation can easily be avoided by performing the check only in an event due to occur later. In the above example, if the ordering check were performed only in the rules associated with f, the violation will be detected as soon as it occurs. Also, the violation will be signaled only once.

At this point is not clear if multiple detection is useful or harmful. However, it is conceivable that certain programs might require multiple detections. The program might wish to know *all* points where the events involved in the error occurred. The above discussion points to having two types of semantics viz., *multiple* and *single* error detection.

Another important issue related to temporal violation is the question of when to set the timer for the binary operators. Based on our current semantics, a temporal violation is valid only if the event number and the value match. Consider the specification

$$\boxed{\textbf{l: occur(e,i,v) wrt c1 before occur(f,j,w) wrt c2 atmost n}}$$

If the ith occurrence of event e had value v, should the timer be set for the jth occurrence of event f even though we are not sure about the value? In the case when the value did happen to be w and we did not set the timer, we would have missed signaling an error early. However it is also possible that the timer was set and it expired signaling an error for which a recovery action was taken. Later when the jth occurrence of event f occurred it was discovered that it did not have value w. Should an 'undo' of the recovery action (now determined to be spurious) be performed or should it be ignored? It is not clear that chosing not to set the timer is any better. In case the jth event never occurred, say due to an infinite loop, we would have missed an opportunity to detect a potential violation.

It appears that these questions can only be answered by considering the needs of an application. Different answers will lead to different languages. One can consider the syntax presented to define 4 languages viz, *multiple, always*, *multiple, certain*, *single, always* and *single, certain*. The difference in the languages is dictated by the semantics. In this thesis we present the transition rules for one of the languages, namely *multiple* detection and *always* setting the timer. The rationale being the programmer can always chose to ignore the types of violations perceived to be spurious by specifying a null handler. To explain other elements in the class, the semantics is a straight forward extension of the one defined here. It is also possible to augment the language with compiler directives which gives the programmer the choice over the semantics. This issue however is not discussed in this thesis.

Rules

In order to identify one temporal checking from another, the transition rules for timing predicates have four dynamic constants viz., event_value, event_type, event_number and event_time with all instances of the **temporal_specification** prototype. TEMPORAL_CHECKER identifies the relevant temporal specifications and sets these variables in them. After this the transition rules are activated. Whenever an event occurrence is detected, a new instance of the relevant transition rules is unfolded with the appropriate parameters. We now explain the meaning of the sixteen types of temporal statements. We explain only the **before** operator. The meaning of the **after** operator is the dual of that for the **before** operator.

The transition rules for all sub cases where * is replaced by* ⊕ <integer constant> and where $ is replaced by $ ⊕ <value_type constant> are not explicitly presented. They can be obtained by substituting event_number by event_number ⊕ <integer constant> and event_value by event_value ⊕ <value_type constant> at the appropriate places. typical case.

To simplify our exposition, let *spec* be a typical instance of **temporal_specification**. The following aliases are assumed. Recall that functions such as FIRST_EV etc., were defined in the section on prototypes.

- FIRST_EV(*spec*) as E and SECOND_EV(*spec*) as F
- FIRST_OC(*spec*) as I and SECOND_OC(*spec*) as J
- FIRST_VAL(*spec*) as V and SECOND_VAL(*spec*) as W
- FIRST_CLK(*spec*) as C1 and SECOND_CLK(*spec*) as C2
- TS_LAB(*spec*) as L and LIMIT_VAL(*spec*) as N

Instead of repeatedly specifying the transition rules to compute the time of the events with every case we define a variable T with the appropriate value. As clock assignment to modules is not specified by the semantics, the transition rules must be sensitive to the case when the clock assigned to a module is different from the clocks in the temporal specifications. Let C be the clock associated with the module in which the specification is defined. Define CURR_LIM to be (REMOTE_TIME_FUNCTION C event_time C1) + N. CURR_LIM computes the interval between the first event with respect to the local clock and the second event with respect to clock C2. Define T to be REMOTE_TIME_FUNCTION C2 CURR_LIM C. T is the interval between the first event and the second event with respect to the local clock. We use the time T to set timers in the relevant cases after the first event has occurred. Similarly, instead of specifying the rules for timing checks with every case, define COND to be '> N' if the LIMIT of the specification is UPPER and to be '< N' if the LIMIT of the specification is LOWER.

In the case when a timing constraint is violated an event of type **temporal_violation** is generated. If the separation was not within specified limits and LIMIT of the specification was LOWER, the value associated with the event is *early.* The value *late* is used in the case when LIMIT is UPPER. For the purposes of the transition rules, let T_ERR_VAL denote the appropriate error condition, i.e., T_ERR_VAL is either *early* or *late.*

Consider $\boxed{\textbf{occur(e,i,v) before occur(f,j,w) atmost } n}$ as an example. In this case COND has value ">n". If the f-event occurs *later* than n units of time after the e-event the value associated with the **temporal_violation** will be *late*. If, in the above example, **atmost** were replaced by **atleast**, COND will have the value "<n" and if the f-event occurs *within* n units of time, the value associated with **temporal_violation** is *early*.

In the transition rules, the timer is set when appropriate only for the **atmost** specification. The transition rules for the **atleast** specification does not require any timer. This is because timers are useful only when there is a known upper bound on when an event should occur. The **atleast** specifies only a lower bound. The upper bound is not specified by it thereby requiring no timer. In this thesis, we discuss specifications involving the **atmost** relation. The transition rules for the **atleast** can be derived from the **atmost** specification by deleting the timer statements (or replacing the timer statements by **null** which does nothing).

We reproduce the skeleton of the specification to help in recalling the syntactic entity for which the semantics are being defined. We use the aliased form (as defined above) with the assumption that all formal use of it will use the actual functions which stand for the alias.

In the following rules, we do not write the **end if** for the **if**'s and use indentation to identify nesting. We omit the limit part of the specification when we present an instance of the **temporal_specification** prototype. We use COND and T_ERR_VAL in the transition rules with the understanding that the correct one will be used. We also omit the clock and the label fields. For example instead of

$$\boxed{\textbf{L: occur(E,*,V) wrt c1 before occur(F,J,\$) wrt c2 atmost N}}$$

we use

$$\boxed{\textbf{occur(E,*,V) before occur(F,J,\$)}}$$

Listed below are the transition rules for all the sixteen cases in complete detail.

<u>Case I</u>: $\boxed{\textbf{occur(E,I,V) before occur(F,J,W)}}$

The E-fired transition rules (i.e., the rules to be executed when an event of type E occurs) are

```
if (event_type = E ) & (event_number = I) & (event_value = V) then
   if GET_EVENT_ENTRY(F,J,W) ≠ ∅ then
      ⟦ generate(temporal_violation, (L,wrong_ordering,E,I,F,J)) ⟧
   else
      SET(F,J,T,L)
```

Notice that as discussed above, we set the timer even though we are not sure that the Jth occurrence of event F will have value W. The corresponding F-fired transition rules are

```
if (event_type = F) & (event_number = J) & (event_value = W) then
   if (GET_EVENT_ENTRY(E,I,V) $\neq$ $\emptyset$) then
      if ( TIME_OF(F,J,C2) - TIME_OF(E,I,C1) COND ) then
         ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,F,J)) ⟧
```

Case II: The E fired transition rule for $\boxed{\textbf{occur(E,I,V) before occur(F,J,\$)}}$ is

```
if (event_type = E) & (event_number = I) & (event_value = V) then
   if GET_EVENT_ENTRY(F,J,) $\neq$ $\emptyset$ then
      ⟦ generate( temporal_violation(L,wrong_ordering, E,I,F,J)) ⟧
   else
      SET(F,J,T,L)
```

The counterpart F fired transition rules is

```
if (event_type = F) & (event_number = J) then
   if (GET_EVENT_ENTRY(E,I,V) $\neq$ $\emptyset$) then
      if ( TIME_OF(F,J,C2) - TIME_OF(E,I,C1) COND ) then
         ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,F,J)) ⟧
```

Case III: $\boxed{\textbf{occur(E,I,V) before occur(F *,\$)}}$

```
if (event_type = E) & (event_number = I) & (event_value = V) then
   if GET_EVENT_ENTRY(F, ,) $\neq$ $\emptyset$ then
      $\forall$ <F,J,W,C,T> $\in$ GET_EVENT_ENTRY(F,,)
         ⟦ generate (temporal_violation(L,wrong_ordering,E,I,F,J)) ⟧
```

Though we use the universal quantifier in the test, it is still executable as any given time the set of occurred events is finite. One could use a for loop to achieve the effect. But using the quantifier simplifies the presentation. Note that in this case we do not set the timer. To be consistent with our universal quantification we do allow null occurrences of event type F. We also signal as many temporal violations as there are F-events. This is to be consistent with the situation where the individual specifications were explicitly specified. The F related transition rules are

```
if (event_type = F) then
   if (GET_EVENT_ENTRY(E,I,V) $\neq$ $\emptyset$) then
      if ( TIME_OF(F,event_number,C2) - TIME_OF(E,I,C1) COND) then
         ⟦ generate (temporal_violation(L, T_ERR_VAL,E,I,
               F,event_number)) ⟧
```

<u>Case IV</u>: There are two sub-cases to consider here, the first being $\boxed{\textbf{occur(E,I,\$) before occur(F,* \$\$)}}$ whose transition rules are defined below.

```
if (event_type = E) & (event_number = I) then
   if GET_EVENT_ENTRY(F,,) ≠ ∅ then
      ∀ <F,J,W,C,T> ∈ GET_EVENT_ENTRY(F,,)
         ⟦ generate (temporal_violation(L,wrong_ordering,E,I,F,J)) ⟧

if (event_type = F) then
   if GET_EVENT_ENTRY(E,I,) ≠ ∅ then
    if TIME_OF(F,event_number,C1) - TIME_OF(E,I,C2) COND then
    ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I, F, event_number)) ⟧

   else
      ⟦ generate (temporal_violation(L,wrong_ordering,E,I,
          F,J)) ⟧
```

The above rule, illustrates the case where a single wrong ordering could result in multiple signals. Let an instance for the second sub-case be $\boxed{\textbf{occur(E,I,\$) before occur(F,*,\$)}}$ the rules for which are

```
if (event_type = E) & (event_number = I) then
   if GET_EVENT_ENTRY(F,,event_value) ≠ ∅ then
      ∀ <F,J,W,C,T> ∈ GET_EVENT_ENTRY(F,,event_value)
         ⟦ generate (temporal_violation(L,wrong_ordering,E,I,F,J)) ⟧

if (event_type = F) then
   if GET_EVENT_ENTRY(E,I,event_value) ≠ ∅ then
      if TIME_OF(F,event_number,c2) - TIME_OF(E,I,c1) COND then
         ⟦ generate (temporal_violation((L, T_ERR_VAL,E,I,
             F,event_number)) ⟧
```

<u>Case V</u>: This case is composed of four sub-cases. Let $\boxed{\textbf{occur(E,*,\$) before occur(F,**,\$\$)}}$ be a representative instance for the first sub-case.

```
if (event_type = E) then
   if GET_EVENT_ENTRY(F,,) ≠ ∅ then
      ∀ <F,J,W,C,T> ∈ GET_EVENT_ENTRY(F,,)
       ⟦ generate (temporal_violation(L,wrong_ordering,E,event_number,
             F,J)) ⟧

if (event_type = F) then
   ∀ <E,I,V,C,T> ∈ GET_EVENT_ENTRY(E, , )
```

```
        if ( TIME_OF(F,event_number,C2) - TIME_OF(E,i,C1) COND) then
            ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,
                F,event_number)) ⟧
```

For the second sub-case consider $\boxed{\textbf{occur(E,*,\$) before occur(F,**,\$)}}$, for which the corresponding transition rules are

```
if (event_type = E) then
    if GET_EVENT_ENTRY(F,,event_value) ≠ ∅ then
        ∀ <F,J,W,C,T> ∈ GET_EVENT_ENTRY(F,,event_value)
            ⟦ generate (temporal_violation(L,wrong_ordering,E,
                event_number,F,J)) ⟧

if (event_type = F) then
    ∀ <E,I,event_value,C,T> ∈ GET_EVENT_ENTRY(E, , event_value)
        if ( TIME_OF(F,event_number,C2) - TIME_OF(E,i,C1) COND) then
            ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,
                F,event_number)) ⟧
```

The transition rules for sub-case three which is $\boxed{\textbf{occur(E,*,\$) before occur(F,*,\$\$)}}$ as follows.

```
if (event_type = E) then
    if GET_EVENT_ENTRY(F,event_number,) ≠ ∅ then
        ⟦ generate (temporal_violation(L,wrong_ordering,E,
            event_number,F,event_number)) ⟧
    else
        SET(F,event_number,T,L)

if (event_type = F) then
    if GET_EVENT_ENTRY(E,event_number,) ≠ ∅ then
        if ( TIME_OF(F,event_number,C2) - TIME_OF(E,event_number,C1)
                COND) then
            ⟦ generate (temporal_violation(L,T_ERR_VAL,E,
                event_number,F, event_number)) ⟧
    else
        ⟦ generate (temporal_violation(L,wrong_ordering,E
            ,event_number,F, event_number)) ⟧
```

The final sub-case is based on $\boxed{\textbf{occur(E,*,\$) before occur(F,*,\$)}}$

```
if (event_type = E) then
    if GET_EVENT_ENTRY(F,event_number,event_value) ≠ ∅ then
        ⟦ generate (temporal_violation(L,wrong_ordering,E,
                event_number, F, event_number)) ⟧
     else
        SET(F,event_number,T,L)

if (event_type = F) then
     if GET_EVENT_ENTRY(E,event_number,event_value) ≠ ∅ then
        if ( TIME_OF(F,event_number,C2) - TIME_OF(E,event_number,C1) COND)
then
        ⟦ generate (temporal_violation(L,T_ERR_VAL,E,
                event_number,F,event_number)) ⟧
```

Case VI: A typical instance of a prototype for this case will be $\boxed{\textbf{occur(E,I,V) before occur(F,*,W)}}$. The e-fired transition rules are

```
if (event_type = E) & (event_number = I) & (event_value = V) then
     if GET_EVENT_ENTRY(F,,W) ≠ ∅ then
        ∀ <F,J,W,C,T> ∈ GET_EVENT_ENTRY(F,,W)
           ⟦ generate (temporal_violation(L,wrong_ordering,E,I,F,J)) ⟧

if (event_type = F) & (event_value = W) then
     if GET_EVENT_ENTRY(E,I,V) ≠ ∅ then
        if ( TIME_OF(F,event_number,C2) - TIME_OF(E,I,C1) COND) then
           ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,
                F,event_number)) ⟧
```

Case VII: Consider $\boxed{\textbf{occur(E,I,\$) before occur(F,*,W)}}$. The transition rules explaining the behavior of the prototype for this case are as follows.

```
if (event_type = E) & (event_number = I) then
     if GET_EVENT_ENTRY(F,,W) ≠ ∅ then
        ∀ <F,J,W,C,T> ∈ GET_EVENT_ENTRY(F,,W)
           ⟦ generate (temporal_violation(L,wrong_ordering,E,I,F,J)) ⟧

if (event_type = F) & (event_value = W) then
     if GET_EVENT_ENTRY(E,I,) ≠ ∅ then
        if ( TIME_OF(F,event_number,C2) - TIME_OF(E,I,C1) COND) then
        ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,F,event_number)) ⟧

     else
        ⟦ generate (temporal_violation(L,wrong_ordering,E,I,F,J)) ⟧
```

Case VIII: Transition rules for this case consist of two sub cases. Let $\boxed{\textbf{occur(E,*,\$) before occur(F,**,W)}}$ be the first sub case. The transition rules for the above specification are as follows

```
if (event_type = E) then
   if GET_EVENT_ENTRY(F,,W) $\neq \emptyset$ then
      $\forall$ <F,J,W,C,T> $\in$ GET_EVENT_ENTRY(F,,W)
         ⟦ generate (temporal_violation)(L,wrong_ordering,E,I,F,J)) ⟧

if (event_type = F) & (event_value = W) then
   $\forall$ <E,I,V,C,T> $\in$ GET_EVENT_ENTRY(E, ,)
      if ( TIME_OF(F,event_number,C2) - TIME_OF(E,k,C1) COND) then
         ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,
              F,event_number)) ⟧
```

The second sub-case deals with the specification $\boxed{\text{occur(E,*,\$) before occur(F,*,W)}}$, the transition rules for which are as follows.

```
if (event_type = E) then
   if GET_EVENT_ENTRY(F,event_number,W) $\neq \emptyset$ then
      ⟦ generate (temporal_violation(L,wrong_ordering,E, event_number,
           F,event_number)) ⟧
   else
      SET(F,event_number,T,L)

if (event_type = F) & (event_value = W) then
   if GET_EVENT_ENTRY(E,event_number,) $\neq \emptyset$ then
      if ( TIME_OF(F,event_number,C2) - TIME_OF(E,event_number,C1) COND)
then
         ⟦ generate (temporal_violation(L,T_ERR_VAL,E,event_number,
              F,event_number)) ⟧
   else
      ⟦ generate (temporal_violation(L,wrong_ordering,E, event_number,
           F,event_number)) ⟧
```

<u>Case IX</u> is represented by the specification $\boxed{\text{occur(E,I,\$) before occur(F,J,W)}}$, the transition rules for which are described below.

```
if (event_type = E) & (event_number = I) then
   if GET_EVENT_ENTRY(F,J,W) $\neq \emptyset$ then
      ⟦ generate (temporal_violation(L,wrong_ordering,E,I,F,J)) ⟧
   else
      SET(F,J,T,L)

if (event_type = F) & (event_value = W) & (event_number = J) then
   if GET_EVENT_ENTRY(E,I,) $\neq \emptyset$ then
      if TIME_OF(F,event_number,C2) - (TIME_OF(E,I,C1) COND) then
      ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,F,J)) ⟧
   else
      ⟦ generate (temporal_violation(L,wrong_ordering,E,I,F,J)) ⟧
```

Case X: The two sub-cases to consider are $\boxed{\textbf{occur(E,I,\$) before occur(F,J,\$\$)}}$ and $\boxed{\textbf{occur(E,I,\$) before occur(F,J,\$)}}$. We shall discuss them one at a time in order. The rules described below describe the operational behavior of the first specification.

```
if (event_type = E) & (event_number = I) then
   if GET_EVENT_ENTRY(F,J,) ≠ ∅ then
      ⟦ generate (temporal_violation(L,wrong_ordering,E,I,F,J)) ⟧
   else
      SET(F,J,T,L)

if (event_type = F) & (event_number = J) then
   if GET_EVENT_ENTRY(E,I,) ≠ ∅ then
      if (TIME_OF(F,event_number,C2) - TIME_OF(E,I,C1) COND) then
      ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,
           F,event_number)) ⟧
   else
      ⟦ generate (temporal_violation(L,wrong_ordering,E,I,F,J)) ⟧
```

The second sub case's behavior is governed by the following transition rules.

```
if (event_type = E) & (event_number = I) then
   if GET_EVENT_ENTRY(F,J,event_value) ≠ ∅ then
      ⟦ generate (temporal_violation(L,wrong_ordering,E,I,F,J)) ⟧
   else
      SET(F,event_number,T,L)

if (event_type = F) & (event_number = J) then
   if GET_EVENT_ENTRY(E,I,event_value) ≠ ∅ then
      if ( TIME_OF(F,event_number,C2) - TIME_OF(E,I,C1) COND) then
         ⟦ generate (temporal_violation((L,T_ERR_VAL,E,I,F,J)) ⟧
```

Case XI: As mentioned in chapter 4, when the temporal constraints were introduced the meaning of this case is the dual of case VI. But for completeness sake we describe the transition rules. A typical element of this case is $\boxed{\textbf{occur(E,*,V) before occur(F,J,W)}}$, the transition rules for which are as follows.

```
if (event_type = E) & (event_value = V) then
   if GET_EVENT_ENTRY(F,J,W) ≠ ∅ then
      ⟦ generate (temporal_violation(L,wrong_ordering,E, event_number,
           F,J)) ⟧
   else
      SET(F,J,T,L)
```

```
if (event_type = F) & (event_number = J) & (event_value = W) then
  ∀ <E,I,V,C,T> ∈ GET_EVENT_ENTRY(E, ,V)
    if ( TIME_OF(F,event_number,C2) - TIME_OF(E,k,C1) COND) then
      ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,
          F,event_number)) ⟧
```

Case XII: The operational meaning described below is for the specification $\boxed{\textbf{occur(E,*,V) before occur(F,J,\$)}}$.

```
if (event_type = E) & (event_value = V) then
  if GET_EVENT_ENTRY(F,J,) ≠ ∅ then
    ⟦ generate (temporal_violation(L,wrong_ordering,E, event_number,
        F,J)) ⟧
  else
    SET(F,J,T,L)
```

```
if (event_type = F) & (event_number = J) then
  ∀ <E,I,V,C,T> ∈ GET_EVENT_ENTRY(E, ,V)
    if ( TIME_OF(F,event_number,C2) - TIME_OF(E,k,C1) COND) then
      ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,F,J)) ⟧
```

Case XIII: This case consists of two sub-cases. Let the first sub case be represented by $\boxed{\textbf{occur(E,*,V) before occur(F,**,W)}}$. The associated transition rules are

```
if (event_type = E) & (event_value = V) then
  ∀ <F,J,W,C,T> ∈ GET_EVENT_ENTRY(F, ,W)
    ⟦ generate (temporal_violation(L,wrong_ordering,E, event_number,
        F,J)) ⟧
```

```
if (event_type = F) & (event_value = W) then
  ∀ <E,I,V,C,T> ∈ GET_EVENT_ENTRY(E, ,V)
    if ( TIME_OF(F,event_number,C2) - TIME_OF(E,k,C1) COND) then
      ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,
          F,event_number) ⟧
```

The second sub-case is represented by $\boxed{\textbf{occur(E,*,V) before occur(F,*,W)}}$, whose transition rules are

```
if (event_type = E) & (event_value = V) then
  if GET_EVENT_ENTRY(F,event_number,W) ≠ ∅ then
    ⟦ generate (temporal_violation(L,wrong_ordering,E, event_number,
        F,event_number)) ⟧
  else
    SET(F,event_number,T,L)
```

```
if (event_type = F) & (event_value = W) then
   if GET_EVENT_ENTRY(E,event_number,V) ≠ ∅ then
      if ( TIME_OF(F,event_number,C2) - TIME_OF(E,event_number,C1) COND)
then
         ⟦ generate (temporal_violation(L,T_ERR_VAL,E,event_number,
              F, event_number)) ⟧
```

Case XIV: Let $\boxed{\textbf{occur(E,*,V) before occur(F,**,\$)}}$ be a typical instance of a prototype for the first sub case. Its transition rules are as follows

```
if (event_type = E) & (event_value = V) then
   ∀ <F,J,W,C,T> ∈ GET_EVENT_ENTRY(F, ,)
      ⟦ generate (temporal_violation(L,wrong_ordering,E,
           event_number,F,J)) ⟧

if (event_type = F) then
   ∀ <E,I,V,C,T> ∈ GET_EVENT_ENTRY(E, ,V)
      if ( TIME_OF(F,event_number,C2) - TIME_OF(E,k,C1) COND) then
         ⟦ generate (temporal_violation(L,T_ERR_VAL,E,i,
              F,event_number)) ⟧
```

The second sub-case is $\boxed{\textbf{occur(E,*,V) before occur(F,*,\$)}}$, whose behavior is explained by

```
if (event_type = E) & (event_value = V) then
   if GET_EVENT_ENTRY(F,event_number,) ≠ ∅ then
      ⟦ generate (temporal_violation(L,wrong_ordering,E,event_number,
           F,event_number)) ⟧
   else
      SET(F,event_number,T)

if (event_type = F) then
   if GET_EVENT_ENTRY(E,event_number,V) ≠ ∅ then
      if ( TIME_OF(F,event_number,C2) - TIME_OF(E,event_number,C1) COND)
then
         ⟦ generate (temporal_violation(L,T_ERR_VAL,E,event_number,
              F, event_number)) ⟧
```

Case XV: The meaning of the specification $\boxed{\textbf{occur(E,*,\$) before occur(F,J,W)}}$ is described by

```
if (event_type = E) then
   if GET_EVENT_ENTRY(F,J,W) ≠ ∅ then
```

```
        ⟦ generate (temporal_violation(L,wrong_ordering,E,
                event_number, F,J)) ⟧
    else
        SET(F,event_number,T,L)

if (event_type = F) & (event_number = J) & (event_value = W) then
    ∀ <E,I,V,C,T> ∈ GET_EVENT_ENTRY(E, , )
        if ( TIME_OF(F,event_number,C2) - TIME_OF(E,I,C1) COND) then
            ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,F,J)) ⟧
```

<u>Case XVI</u>: The final case consists of two sub-cases viz. $\boxed{\text{occur(E,*,\$) before occur(F,J,\$\$)}}$ and $\boxed{\text{occur(E,*,\$) before occur(F,J,\$)}}$, whose effect is described by the following transition rules.

```
if (event_type = E) then
    if GET_EVENT_ENTRY(F,J,) ≠ ∅ then
        ⟦ generate (temporal_violation(L,wrong_ordering,E, event_number,
                F,J) ⟧
    else
        SET(F,J,T)

if (event_type = F) & (event_number = J) then
    ∀ <E,I,V,C,T> ∈ GET_EVENT_ENTRY(E, ,)
        if ( TIME_OF(F,event_number,C2) - TIME_OF(E,I,C1) COND) then
            ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I,F,J) ⟧
```

The meaning for the second sub-case is given by the following rules

```
if (event_type = E) then
    if GET_EVENT_ENTRY(F,J,event_value) ≠ ∅ then
        ⟦ generate (temporal_violation(L,wrong_ordering,E, event_number,
                F,J) ⟧
    else
        SET(F,J,T,L)

if (event_type = F) & (event_number = J) then
    ∀ <E,I,V,C,T> ∈ GET_EVENT_ENTRY(E,i,event_value)
        if ( TIME_OF(F,event_number,C2) - TIME_OF(E,i,C1) COND) then
            ⟦ generate (temporal_violation(L,T_ERR_VAL,E,i,F,J) ⟧
```

This concludes the transition rules for the binary usage of the occur relation. We now turn our attention to the formal semantics of the unary temporal statements. Recall, that these statements required something to happen i.e., they are similar to liveness requirements.

Case A: The transition rules describing the effect of $\boxed{\textbf{occur(E,I,V)}}$ is given by

```
if (event_type = E) & (event_number = I) then
   if (event_value ≠ V) then
      ⟦ generate (temporal_violation(L,wrong_value,E,I,
               event_value,V)) ⟧
   elsif [TIME_OF(E,I,C1) COND] then
      ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I)) ⟧
```

Case B: The following is the transition rules of the specification $\boxed{\text{occur(E,*,V)}}$, which requires all events of type E to have value V.

```
if (event_type = E) then
   if (event_value ≠ V) then
      ⟦ generate (temporal_violation(L,wrong_value,E, event_number,
               event_value,V) ⟧
   else if [TIME_OF(E,event_number,C1) COND] then
      ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I)) ⟧
```

Case C: The transition rules which explain $\boxed{\text{occur(e,i,\$)}}$ are

```
if (event_type = E) & (event_number = I) then
   if [TIME_OF(E,event_number,C1) COND] then
      ⟦ generate (temporal_violation(L,T_ERR_VAL,E,I)) ⟧
```

Case C: The transition rules explaining $\boxed{\text{occur(e,*,\$)}}$ are

```
if (event_type = E) then
   if [TIME_OF(E,event_number,C1) COND] then
      ⟦ generate (temporal_violation(L,T_ERR_VAL,E,event_number)) ⟧
```

5.6.1 Interval Semantics

In this section a variation of the semantics for the temporal ordering statements is provided. Recall that the semantics we defined above involved only point time. The point time was obtained by using an averaging function called AVERAGE, which was a parameter to the semantics. In certain situations, the averaging function may not be very appropriate. This is true in systems where the range of error that can

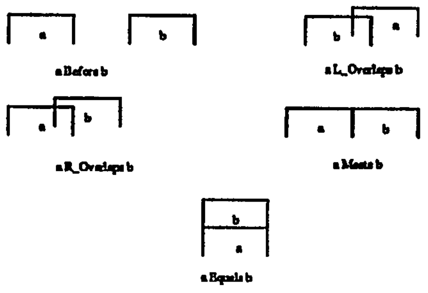


Figure 5.4. Possible relations between intervals

be tolerated is very small. The information lost in applying the averaging function might be significant for certain applications.

We present a meaning to the temporal ordering functions which is based on an interval definition of time. As mentioned before, interval time does not affect the discussion of time and its relation with multiple clocks. Only the meaning of the temporal operators **before** and **after** has to be altered.

In the point definition of time two times were either equal or one was less than the other. An event 'a' is said to be before another event 'b' if the numerical value associated with 'a' is less than the numerical value associated with 'b'. In the interval case, given two intervals there are six possible relations between them. This is shown in figure 5.4.

Before we describe the meaning of the timing constraints, more values for the predefined event **temporal_violations** are introduced. The only values for the point time case were **wrong_ordering**, **early** and **late**. In the interval semantics, however, more values are needed to characterize the types of intervals. The new values are **r_overlaps**, **l_overlaps**, **during**, **equals**, **meets**, **guaranteed** and **possible**. The last two values are relevant only when the intervals are disjoint. An event with the **guaranteed** value indicates that irrespective of when exactly the relevant events occurred, the constraint was violated. The value of **possible** indicates that depending on when exactly the events occurred, a violation is possible. However, in certain cases violation may not have occurred but our information regarding time is not accurate enough to certify this.

Rather than discuss the meaning of each of the temporal specifications individually, we present two examples. The meaning for the other cases can be derived from these examples. The first example has the limit as **atmost** while the second has the limit as **atleast**. Let

$\boxed{\text{l: \textbf{occur(E,I,V) wrt c1 before occur(F,J,W) wrt c2 atmost} n}}$

be the first example of a temporal specification. To further simplify matters, the rules regarding relevancy of the specification is not reproduced. The following transition rules are executed when checking the actual timing constraint.

Let [min1,max1] be TIME_OF(E,I,C1) and [min2,max2] be TIME_OF(F,J,C2) The following rule checks for satisfaction.

```
if (min1 < min2 < max1 ≤ max2) then
   ⟦ generate(temporal_violation (L,r_overlaps,E,I,F,J)) ⟧
elsif (min2 < min1 < max2 ≤ max1) then
   ⟦ generate(temporal_violation (L,l_overlaps,E,I,F,J)) ⟧
elsif [( min1 < min2) & (max2 < max1)] ∨ [(min2 < min1) & (max1 < max2)]
then
   ⟦ generate(temporal_violation (L, during,E,I,F,J)) ⟧
elsif [(min1 = min2) & (max1 = max2)] then
   generate(temporal_violation (L, equals,E,I,F,J)) ⟧
elsif (max1 = min2) then
   ⟦ generate(temporal_violation (L, meets,E,I,F,J)) ⟧
elsif (max1 < min2) then
   if [(max2 - min1) > n] then
      ⟦ generate(temporal_violation (L, possible,E,I,F,J)) ⟧
   elsif [ (min2 - max1) > n] then
      ⟦ generate(temporal_violation (L, guaranteed,E,I,F,J)) ⟧
   endif
end if
```

The meaning of temporal constraints involving **atleast** is as follows. As above, let [min1,max1] be TIME_OF(E,I,C1) and [min2,max2] be TIME_OF(F,J,C2).

```
if (min1 < min2 < max1 ≤ max2) then
   ⟦ generate(temporal_violation (L, r_overlaps,E,I,F,J)) ⟧
elsif (min2 < min1 < max2 ≤ max1) then
   ⟦ generate(temporal_violation (L, l_overlaps,E,I,F,J)) ⟧
elsif [( min1 < min2) & (max2 < max1)] ∨ [(min2 < min1) & (max1 < max2)]
then
   ⟦ generate(temporal_violation (L, during,E,I,F,J)) ⟧
elsif [(min1 = min2) & (max1 = max2)] then
   ⟦ generate(temporal_violation (L, equals,E,I,F,J)) ⟧
elsif (max1 = min2) then
   ⟦ generate(temporal_violation (L, meets,E,I,F,J)) ⟧
elsif (max1 < min2) then
   if [(max2 - min1) > n] then
      ⟦ generate(temporal_violation (L, guaranteed,E,I,F,J)) ⟧
```

```
    elsif [ (min2 - max1) > n] then
        ⟦ generate(temporal_violation (L, possible,E,I,F,J))⟧
    endif
endif
```

This concludes the discussion of interval semantics, as rewriting the transition rules for all the temporal constraints is a straightforward exercise.

This completes our discussion of the transition rules for the temporal specifications permitted by the language. Further topics such as temporal reasoning about programs should be based on these rules but are not relevant to the semantics per se.

5.6.2 Transition Rules for Subprograms

As subroutines are the principal executable units in a program, the dynamic semantics of the subroutine determines the execution of the program. As discussed in the initial configuration, there is a specific automata executing a class of subroutines. The characteristic of these automata are presented in this section terms of a dynamic algebra. The various states that they evolve through are categorized into a two 'super' states. These are distinct from the state of the program.

Recall that in our effort to retain the advantages of the various paradigms of programming, we classified subprograms into functions, observers and procedures. The transition rules for each of these classes differ slightly. Functions are to be evaluated lazily unless defined using **efunction** and parallel invocations can be simultaneously active. Multiple invocations of observers can be executed in parallel unless restricted by exclusive access lists. However they cannot be executed lazily. As they have access to mutable variables they could return different results at different times. The semantics of procedures disallows simultaneous/parallel threads of procedures/observers in the same exclusive access list and cannot be evaluated lazily.

Define STATES, the set of 'super' states that a subprogram machine could be in, to be { *ready*, *busy* }. A automaton which represents a set of subprograms in a *ready* state can accept an invocation from an event occurrence, the right hand side of a causal statement or a periodic task, if there is an available thread. When it accepts a call, the state changes to *busy*, a state in which the parameters are evaluated, the body executed.

When the state of a procedure is *busy*, the state of all other subprograms in its exclusive access class is implicitly changed to *busy* as we have only one automaton per equivalence class. Hence a sequence of procedure invocations is like a monitor [43] but without the syntactic baggage associated with them. When the end of the procedure is reached, it changes the state of the corresponding automata to *ready*, which also causes an implicit change in the state of automata in its equivalence class to *ready*.

Functions and observers operate differently. We allow parallel executions of multiple invocations of the same function. In order to be realistic, there is a bound on the number of parallel invocations. This causes functions to reach the state *busy* only when all permissible threads are active. A parametric constant THREADS indicates the maximum number of active threads. A dynamic constant CURR_ACTIVE keeps track of the number of active threads which was assigned to 0 in the initial structure.

The start of every thread increments the value of CURR_ACTIVE by one. If the value of CURR_ACTIVE is equal to the constant THREADS, no new thread can be activated. In this state all subsequent subprogram calls are blocked till a thread is freed. On return the value of CURR_ACTIVE is decremented by 1.

Since functions can be executed lazily, they are compiled into *combinators.* As explained in chapter 3, lazy evaluation is automatically supported by *combinators.* Therefore, the transition rules for functions will be the reduction rules associated with the combinators.

Notice that the above discussion applies only to subprograms which are the start of the thread. An active thread can activate all subprograms it requires and we do not specify a maximum depth of subroutine calls. The maximum depth of subprogram calls can be specified as resource restrictions.

To formalize the above stated conditions in the form of transition rules, define the *core meaning* of a procedure call as evaluating the parameters, executing the body and returning to the called unit. For a subprogram q, denote the core meaning of a call to it by $Core(\ [\![\ \mathrm{q(PARAMS(q))}\ ]\!]\ )$. It is to be interpreted as executing q with the parameters of q. The core meaning of a subprogram call does not involve state or thread checking. We do not discuss the transition rules for the core meaning, as for eager functions, observers and procedures it is a straightforward modification of the discussion in [65], and for functions it is a modification of the combinator method described in section 3.2.5.

Consider p $\in$ PROCEDURES i.e., an instance of the **subprograms** prototype with effect class *procedure* and q $\in$ **Subprograms** such that q is callable from p. In this case there is no new thread started. Therefore the semantics of the call is as usual. When call to q becomes the current executable statement in p the following transitions rule is executed

$$Core(\ [\![\ \mathrm{q(PARAMS(q))}\ ]\!]\ )$$

We discuss the activation of new threads. Note that threads are synonymous with subprogram calls activated by event generation, periodic tasks and causal statements. Recall that these statements queued their request in a structure operated by ADD_TO_LIST and REMOVE_FROM_LIST. The semantics for the subprogram involved dequeuing a request via the REMOVE_FROM_LIST operation. To start a thread based on the execution of a procedure p the following rules are executed.

```
if (p.state = ready) & (CURR_ACTIVE < THREADS) then
   p.state := busy ;
   CURR_ACTIVE := CURR_ACTIVE + 1;
   REMOVE_FROM_LIST( Q_LAB(MY_SELF),params, REPLY_TO,
                                      WHO_CALLED) ;
   Core ⟦ p(params) ⟧
```

In the above rule, if the automaton is ready and a thread of execution is available, the automaton becomes busy, extracts the parameters from its queue and executes the body. If p were an element of **Functions** or **Observers** not mentioned in any exclusive access lists, the transition rules that will be executed are

```
if (p.state = ready) & (CURR_ACTIVE < THREADS) then
   CURR_ACTIVE := CURR_ACTIVE + 1;
   p.LOCAL_THREADS := p.LOCAL_THREADS - 1;
   if (p.LOCAL_THREADS = 0) then
      p.state := busy;
   end if
   REMOVE_FROM_LIST( Q_LAB(MY_SELF),params, REPLY_TO,
                                      WHO_CALLED);
   Core ⟦ p(params) ⟧
```

In this case, also the automaton has to be ready. If it is, it extracts the parameters and executes the body. However, the automaton becomes busy only if the maximum permissible number of concurrent threads has been attained.

The meaning of the **return** statement is now presented. If a **return** statement is not explicitly present the end of the routine can be treated as a **return** statement. The first action taken is to decrement CURR_ACTIVE and change the state to *ready.* If the automaton is a function or an observer not in any exclusive access list, its LOCAL_THREADS is incremented. The other action concerns the semantics of the **causal** statement. The event handler state has to be set appropriately, and if the right hand side of the **causal** statement returned false an event of type **disaster** has to be generated.

These actions involve a new dynamic constant VAL_RET which has the value to be returned. The core meaning of the subprogram assigns a value to VAL_RET. If no value is to be returned, VAL_RET is undefined ($\perp$).The rules are

```
if WHO_CALLED ∈ EVENT_LABELS then
  if VAL_RET = false then
    EVHAN_STATE := EVHAN_STATE′
  end if
elsif WHO_CALLED ∈ CAU_LABELS then
  if VAL_RET = false then
    ⟦ generate(disaster, MY_MODULE) ⟧
  end if
end if
```

Recall that EVHAN_STATE was a dynamic function indicating the activeness or passiveness of an event handler. In the case of the subprogram returning false, the state of the handler must be made passive. In the above rule, EVHAN_STATE′ is the new function derived from EVHAN_STATE. Its behavior is defined as

$$\text{EVHAN\_STATE}'(\text{el}) = \begin{cases} \textbf{passive} & \text{if el} = \text{WHO\_CALLED} \\ \textbf{active} & \text{otherwise} \end{cases}$$

5.7 Periodic Tasks

The transition rules for a periodic task (subprogram calls at fixed time intervals), is presented. It has a dynamic constant named LAST_TIME which denotes the last time the task was scheduled. As the task's periodicity is specified with respect to a clock which is not necessarily the clock assigned to the module in which the task is declared, it is necessary to calculate the time with respect to the local clock against the specified clock. This is stored in the variable 'temp_time'. If the difference between temp_time and LAST_TIME is greater than or equal to the specified period, the task is activated. The activation is represented by queuing it on the relevant subprogram using an ADD_TO_LIST. In the transition rule described below, let *parameters* represent the evaluation of the **expr** projection of PT_PARAM(MY_SELF), i.e., the parameters to the periodic task. Also recall that PT_NAME(MY_SELF) returns the name of the subprogram specified in the periodic task.

```
temp_time := REMOTE_TIME_FUNCTION (MY_CLOCK,
                current_time, PT_CLK)
if ( temp_time - PTLAB(MY_SELF).LAST_TIME) ≥
              PT_INTERVAL(MY_SELF) then
  ADD_TO_LIST(Q_LAB(PT_NAME(MY_SELF)), (parameters,⊥,
                    PTLAB(MY_SELF)))
  PTLAB(MY_SELF).LAST_TIME := temp_time
end if
```

The above rule by itself is not sufficient. It does not prevent a scheduler from executing the rule only every nth (n > 1) interval. If one has to guarantee the scheduling every interval, at the semantic level one must add the scheduling as an axiom. This can be done in a fashion similar to the axiom governing the environment. It is not obvious that is it appropriate to specify that an interval violation should neve occur. Certain situations might require the postponing of the periodic task for something determined to be more urgent. As a consequence, an event of type **periodic_violation** is generated whenever the task is not scheduled within the specified period. To effect this is the following check is made in the above transition rule.

```
if ((current_time - PTLAB(MY_SELF).LAST_TIME) div PT_INTERVAL>1) &
    ( PTLAB(MY_SELF).LAST_TIME > PTLAB(MY_SELF).ERR_TIME)
            then
  PTLAB(MY_SELF).ERR_TIME := current_time
  ⟦ generate(periodic_violation,PT_NAME) ⟧
end if
```

The antecedent of the above rule has two components to it. The first checks if the elapsed time is greater than the period associated with the task. The second component ensures that only one violation is indicated for the non-scheduling of the task. This check is performed by comparing the time when an error was signaled (ERR_TIME) and the time of last scheduling (LAST_TIME). If both the conditions are satisfied, the time of signaling the error is updated and an error signaled.

The above described semantics can be better achieved using timers. That is the execution of periodic tasks can be made more efficient by queuing up on a periodic task timer. However in the semantics the timers were event based. Instead of defining a new type of timer, we opted for the simpler definition. An implementation can choose to activate the above transition rule when a timer expires thus improving the efficiency.

5.8 Distributed Semantics

Having defined all the relevant transition rules for ARL, we justify our claim that ARL and its semantics do constitute a distributed language. We have characterized a distributed sytem using multiple clocks and permitting the mapping of clocks to modules. Described below is a precise characterization of 'distributed semantics'.

Define a function CLK_ASSIGN captures the mapping of a clock to modules. That is, CLK_ASSIGN can be typed as MODULES $\rightarrow \mathcal{P}$(CLOCKS). Note that if a map for a module is specified then CLK_ASSIGN for that module will have only one element. That is if **for** M **use** C is present in the definition of the module, CLK_ASSIGN(M) = { C }. A module can be assigned more than one clock if a specific map is not defined. This is to be interpreted as capturing all permissible implementations.

Define a MODULES indexed function TSPEC_CLK which assigns a clock to a temporal specification in the module. TSPEC_CLK can be typed as TEMPORAL_LABELS → CLOCKS. From this definition it is clear that we permit only one clock to be assigned to a given temporal specification. Therefore, a temporal specification is an indivisible unit. Clearly, the clock that is assigned to a temporal specification must be one of the clocks that was assigned to the module. Hence the following constraint must be satisfied by TSPEC_CLK:

```
if M.TSPEC_CLK(tlab) = C then
   C ∈ CLK_ASSIGN(M)
```

To specify the mapping of a clock to other elements define a function REST_CLK: NODES → CLOCKS. REST_CLK satisfies the same constraint as TSPEC_CLK. The reason for defining REST_CLK separately from TSPEC_CLK is only to differentiate between temporal constraints and other elements.

Note that the functions defined above are non-trivial only if a mapping of clocks to modules is not completely defined. The distributed semantics of a given program is the operational semantics (as defined by the transition rules) indexed by the three functions defined above.

5.9 Summary

This concludes the development of the operational semantics of the language. At first the meaning of evolution using the transition rules of the ARL machine was developed. The initial structure of an ARL program along with assumptions about the environment was presented. Finally, the transition rules for the syntactic elements in ARL was presented. Appendix B has the list of all the functions used in the semantics. Examples of programs in ARL is presented in the next chapter.

CHAPTER 6

EXAMPLES

One of the principal goals of this research was to design a language that was to be used in prototyping systems. Expressiveness is the key concern of prototyping languages and attention was paid to this in the design of ARL. In this chapter we demonstrate the expressiveness of ARL by programming systems whose specifications are found in the literature. Note that we use an informal technique as there is no formal measure of the expressiveness of a language. Our approach to show that the language has expressiveness consists of two components. Both the components actually develop programs or program fragments in ARL. The first component deals with specifications for problems discussed in the literature in ARL, while the second is about comparing ARL with other languages.

Initially, specifications to certain problems discussed in the literature are discussed. The first problem consists of two related sub-problems. These are two basic problems in real-time computation and solutions for them are presented. The first sub-problem is to specify a tight control loop, while the second sub-problem is to specify a jitter free computation. The other problems selected from the literature are 1) the dining philosophers problem [70] 2) specifying a train system [61] and 3) Semantics of the timed entry call in Ada [1].

The second component compares ARL with other existing languages. We use ESTEREL [10] and real-time Prolog [30] as two languages to compare with ARL. A program in ESTEREL is presented first followed by an equivalent ARL program. Following this, the differences between the two approaches is discussed. Similarly, a real-time Prolog program is followed by the ARL program and their comparison.

6.1 Solutions to Two Basic Problems

ARL solutions to two common problems in real-time computation are presented. Both these problems are in the context of periodic tasks. The first problem concerns the specification of a tight control loop. That is, the time for the control loop must be as small as possible. This is to be interpreted by the scheduler (defined by the implementation) as a non-preemptable section of the task. Such a case is shown in figure 6.1. In the figure, 'start' and 'finish' indicate the start and finish of the control loop (the non-preemptable section of the task) and not of the task itself. The task has to start and finish within the period shown.

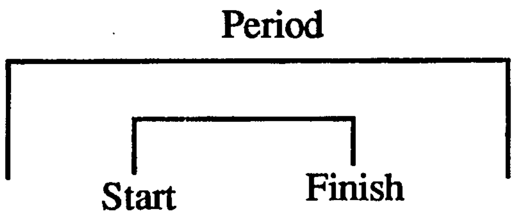


Figure 6.1. Limit Case

The second problem is also in the context of a periodically executed task. In this case the duration of the control loop is not crucial. This gives the scheduler a certain amount of freedom in when to schedule and preempt the task within the period. However the task performs a specific function which must be performed at a fixed interval i.e., relatively jitter free. Such a case is shown in figure 6.2. In this case 'start' and 'finish' could denote the start and finish of some time critical section of the periodic task, while 'sampling' denotes the point where the sampling actually takes place.

6.1.1 Limit

In this paragraph, we show how to specify temporal constraints for the tight scheduling case. The task which is to be executed periodically is augmented with

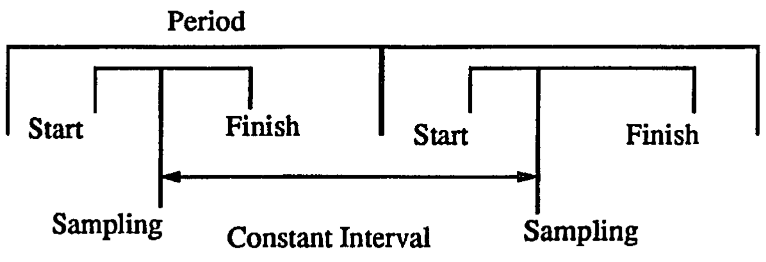


Figure 6.2. Fixed Offset

two events 'start' and 'finish'. Refer to figure 6.1. The 'start' event indicates the start of the tight control loop, while the 'finish' event indicates the finish of the loop. Let the task be called 'periodic_task' and let there be only one clock in the system.

```
    module limit is
        auto (cl := cl + 1) init 0
        auto (cl >> T) periodic_task
{Periodic task scheduled for execution every T units of time}
        event start, finish

        procedure periodic_task is
        begin
            { Body }
            ⋮
            generate(start)
            ⋮
            generate(finish)
            ⋮
        end
        { Temporal Constraint }
        occur(start,*,$) wrt cl before occur(finish,*,$$) wrt cl atmost ε1
    end limit;
```

In the temporal specification above, ϵ_1 represents the maximum permissible time the sub-task can take. Note that in the above specification there is no minimum time specified. One could add such a specification using the **atleast** operator.

6.1.2 Jitter Control

The following is a solution to the second problem discussed above. Refer to figure 6.2. The example program for this case is almost identical as above. The main difference is that only one event is used to indicate the offset and is generated at the sampling point. Call that event 'signal.' This event will be generated somewhere between the generation of 'start' and 'finish' (i.e. within the critical section.) Let the error margins which control the jitter in scheduling and execution of the task be (sig_mar_1 and sig_mar_2). The following requirements can be augmented with bounds on 'start' and 'finish' as shown above. The skeleton of the periodic task and the temporal constraints for the jitter control are as follows.

```
procedure periodic_task is
begin
```

```
        { Body }
        ⋮
        generate(start)
        ⋮
        generate(signal)
        ⋮
        generate(finish)
        ⋮
end
occur(signal,*,$) wrt cl before occur(signal,*+1,$$) wrt cl
                                 atmost(T + error_sig_mar_1)

occur(signal,*,$) wrt cl before occur(signal,*+1,$$) wrt cl
                                 atleast(T - error_sig_mar_2)
```

6.2 Dining Philosophers

The following program illustrates the use of the functional aspect of the language as a major part of the solution for a concurrent problem. We chose the dining philosophers problem [70] to show how a solution in the purely functional language (Miranda) can be specified. This solution is a direct translation of the algorithm in [70] where at no time does a philosopher wait with just one fork. It is possible to code the waiting with one fork case in Miranda but it complicates the program. The scheduling of when a philosopher wants to eat, finish eating etc. is not specified. The Miranda (ARL) program is as follows.

```
{ Number of philosophers }
num_phil = 5

{ The various states that a Philosopher can be is defined below. The state of a
computation is a sequence (of 'num_phil' elements) consisting of the state of the
philosophers in the system }
phil_state :: Hungry | Eating | Waiting | Thinking

{ nth is a function that has a sequence x and an integer i as its argument. It
returns the ith value in the sequence. }
nth x i = x!i

{ from is a function similar to nth. It has three arguments a sequence x, i and
j. It returns a subsequence starting at i and ending at j. }
```

```
from x i j = [x!n | n<-[i ..j] ]

{ Test determines whether the required forks for philosopher i in state 'state' are
free}

test state i = True ,
                     (nth state ((i -1) mod (num_phil-1)) ~= Eating) &
                     (nth state i = Hungry) &
                     (nth state ((i + 1) mod (num_phil-1)) ~= Eating)
             = False, otherwise

{ Pickup returns a state after philosopher i has requested the forks.}

pickup state i = (from state 0 (i-1)) ++ [Eating] ++ (from state (i+1) (num_phil-1)) ,
                                        test (from state 0 (i -1) ++ [Hungry]
                                        ++ from state (i+1) (num_phil-1)) i
{ test failed hence wait }
               = from state 0 (i -1) ++ [Waiting] ++
                 from state (i+1) (num_phil-1), otherwise

{ Philosopher i has finished eating. Wake up a waiting neighbor if any}

putdown state i = pickup new_state prev, nth new_state prev = Waiting
                = pickup new_state next, nth new_state next = Waiting
                = new_state, otherwise
                          where
                          new_state = from state 0 (i -1) ++ [Thinking]
                                      ++ from state (i+1) (num_phil-1)
                          next = (i+1)mod(num_phil-1)
                          prev = (i-1) mod(num_phil-1)
```

As the program is in a purely functional language, the state of the computation is passed as an argument to all the functions, which return the modified state as the result. For this example it is clear that the functions 'pickup' and 'putdown' should *not* be executed concurrently. This can be specified in ARL by using the exclusive access lists as '**excl**{pickup, putdown}'.

6.3 A Train System

The problem presented in this section, was originally described in [61]. The principal concern is to specify a railway system consisting of a network of tracks on which an arbitrary number of trains may run asynchronously. The restrictions are 1) Only one train may occupy a particular track. 2) If the track required by a train is occupied it waits for it on its current track, i.e., there is no buffer zone between the tracks. 3) All solutions must be independent of the number of trains/tracks in the system. 4) There is an initial track assignment to each train in the system.

The input to the system will be the layout of the network and the route various trains take. The input will be assumed to be consistent and hence no checking for validity is performed. We add temporal constraints regarding the waiting for tracks and occupancy of tracks. We assume that the route of each train is given as a sequence of track numbers.

The general structure of our solution is as follows. Each train is considered to be a separate module having knowledge of the path it requires. But as the behavior of all the trains is similar, we use a parametrized module to denote them. The assignment of trains to tracks is maintained by a module called 'resource_manager.' A train requests the next track it requires by generating an event of type 'want_track.' If the request can be granted (i.e., there is no train on the requested track), 'resource_manager' allocates the track to the train and informs the train via an event of type 'ack'. In the case that the requested cannot be granted, the train is said to be blocked. The set of blocked trains and the tracks on which they are blocked is maintained by a module called 'task_control'. 'task_control' is informed by 'resource_manager' about the blocking of a train via an event of type 'wait_track'. When a track becomes available, a message to activate a waiting train (if any) is sent to 'task_control' from 'resource_manager'. Note that there is no main program per se as the trains by executing their initialization code (requesting the first track) activate the whole program.

The system described here is a single clock system. It can be extended to a multi-clock system with each train, the resource_manager and task_control having their own clock.

```
type_pool t1 is
 track_assignment :: (integer, integer)
{ A tuple representing the track number and the train number }
end t1

with t1
event_pool e1 is
{ Definition of event types }
 event want_track :: track_assignment
 event wait_track :: track_assignment
 event wake_up :: integer
 event ack :: integer
end e1

{ Clock definition }
```

```
clock_pool cl is
  auto (timerl := timerl + 1) init 0
end cl

with el,cl,tl
module resource_manager is
{ This module is in charge of assigning tracks to trains }

{ Variable Declaration }
track_assign: [integer] {assignment of tracks to train. A value of -1 indicates the
track of the index is free }
train_assign: [integer] {assignment of trains to track.}

{ Procedure update is invoked when an event of type want_track occurs. As want_track
does not signify a fault, the right hand side of the causal statement can be null }
  update(want_track) causes null

  procedure update (track_no,train_no) return boolean is
{ Note the use of pattern matching and lack of parameter type declaration }
    old_track : integer;   begin
    if track_assign[track_no] = -1 then
{ Track is free }
      old_track := train_assign[train_no]
      track_assign[old_track] := -1 || Free old track
      train_assign[train_no] := track_no
      track_assign[track_no] := train_no
      reply(ack(train_no)) { train can continue}
{ The delay below is to wait for time to let train leave the track. This can be better
achieved by a sequence of events but it complicates the example }
      delay move_time
{ Wake up any train waiting for the track }
      generate(wake_up(old_track))
    else
{ Track not free. Wait for it }
      generate(wait_track(track_no,train_no))
    end if
    return true{ to avoid becoming passive}
  end { update }

end resource_manager

{ The train system is defined as a parametrized module. This is done as the behav-
iors of all the trains are similar }
with el,cl
```

```
module train_system [train_no : 1 .. N]

  path : [integer]
{ Sequence of integers indicating path of the train}
  next : integer
{ Signifies the index of the track the train is go next. }

{ Event Handler: When an ack is received, can continue }
  check(ack) causes null

  procedure check(who_to) return boolean is
{ Note the requirement of type inferencing for parameters also}
  begin
    if (who_to = train_no) then
    { If relevant go to next track }
      next := next + 1;
{ Let some_time(i) represent the time it occupies the ith track }
      delay some_time(next - 1);
{ Request the new track }
      generate(want_track(path[next],train_no))
    end if
  return true
  end { check }

{ Timing Constraint: Let limit(i) represent the limit for train i. It has to be a
compile time constant in keeping with the definition of the language. }
occur(want_track,*,$) before occur(ack,*,$$) atmost limit(train_no).

{ The above specification assumes, that each train has the an identical upper limit
for all the tracks in its journey. If not, a different timing requirement for each track
segment has to be specified. }

init{ The initialization of the train module }
  path := get_path(train_no)
  next := 0;
  generate(want_track(path[next], train_no))
end train_system

with el,tl
module task_control
{ Controls the activation of waiting trains }

waiting : [track_assignment]
{ Sequence of waiting trains along with the tracks for which they are waiting}
```

```
  sleep(wait_track) causes null
  awaken(wake_up) causes null
{ Event Handlers: The event wait_track causes a train to be added to waiting, while
the event wake_up causes a train waiting for the specified track to be awakened. }

{ Sleep is invoked to add a train to the waiting list }
  procedure sleep(track_no, train_no) return boolean is
  begin
    waiting := waiting ++ [(track_no,train_no)]
    return true
  end { sleep }

{ when a track becomes free, awaken is called. A train waiting for the track is
activated }
  procedure awaken(track_no) return boolean is
  begin
{ Select the first train which is waiting on track_no. The Z-F notation is used. If
the expression is evaluated lazily, the iteration will terminate when the first element
is found. }
    chosen := [(x,y) | (x,y)<-waiting, x=track_no]!0
    waiting := waiting -- [chosen] { delete the chosen element}
{ Inform the resource_manager about the activation. }
    generate(want_track(chosen))
    return true; || To avoid becoming passive   end { awaken }

{ This module has no initialization code to be executed. }
end task_control
```

6.4 Timed Entry Call in Ada

The examples discussed above did not use the semantics associated with multiple clocks in ARL. In this section, we discuss possible semantics of the timed entry call in Ada [1] using ARL. The timed entry call in Ada specifies an upper-bound on the time it takes to accept an entry call. Consider the following example.

```
select
     SERVER.FAST_CALL(p);
-- an entry call which has as associated deadline
or
     delay D; -- time limit for accepting the call
     ALTERNATE;
end select
```

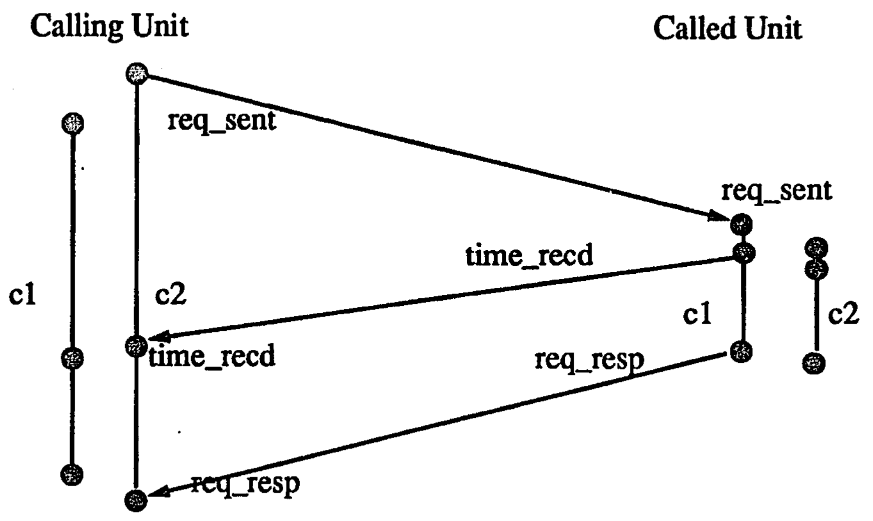


Figure 6.3. Request and Response for Timed Entry Call

The unit issuing the timed entry call will wait for D units of time for the entry call to be accepted. If the entry call cannot be accepted with that time, the entry call is canceled and the task executes 'ALTERNATE'. We discuss the effect of various ways of measuring time on the entry call. This is important when such calls are made in a distributed setting. What follows is a summary of suggestions made in [94] translated into ARL.

We represent the issuing of the entry call by an event 'req_sent'. The time within which the call is to be accepted is the value associated with the event. The called task responds with an event 'req_resp'. If it can accept the call the enumerated value 'accepted' is associated with the event. Otherwise the value 'rejected' is sent. Assume that the system has two clocks c1 and c2. Also assume that one of the clocks is associated with the calling site while the other is associated with the called site. All event occurrences can be measured with respect to either of the clocks. For example, the occurrence of 'req_sent' can be measured with respect to either c1 or c2; similarly for 'req_resp'. It is possible to generate an event indicating the time at which an event occurred. This new event conveys the time of occurrence of an event at a different module. For the purposes of this example let an event called 'time_recd' be generated by the called unit just after it receives 'req_sent'. All the possible variations are shown in figure 6.3.

Recall that in a multi-clock environment, the semantics created a map of clocks as viewed from one of the clocks. For example, let C1, C2 and C3 be the clocks in the system. Associated with C1 is a table which contains an approximate value for clocks C2 and C3 for certain values of C1. Similarly for C2 and C3. Thus, when a reference to a potentially remote clock is made, a message is not dispatched, but rather the clock's map is consulted.

We discuss various temporal specifications that can be stated using the two events and clocks. First we consider specifications on the calling unit. The first possible temporal specification is

$\boxed{\textbf{occur}(\text{req\_sent},*,\$)\ \textbf{wrt}\ \text{c1}\ \textbf{before}\ \textbf{occur}(\text{req\_resp},*,\$\$)\ \textbf{wrt}\ \text{c1}\ \textbf{atmost}\ \text{D}}$

Here the maximum time the module can wait after issuing a request before it receives an acknowledgment regarding the request's acceptance is equal to the deadline in the original task. If acknowledgment is not received, a timer will expire. The handler for timer expiration will then execute the alternate action. We present the causal statement along with an equational specification of the handler for the timer case. Let the temporal specification be labeled '1' and x represent the instance number of 'req_resp' in question. The ARL specification will be as follows.

FALSE(**temporal_violation**) **causes** handler
handler (1, **timer_expired**,req_resp,x) = ALTERNATE; **return true**

The specification described above can be modified to

$\boxed{\textbf{occur}(\text{req\_sent},*,\$)\ \textbf{wrt}\ \text{c1}\ \textbf{before}\ \textbf{occur}(\text{req\_resp})\ \textbf{wrt}\ \text{c2}\ \textbf{atmost}\ \text{D}}$

In this case, the error in the map of the relevant clock is also taken into account. In both these cases, one could use either the point or the interval nature of time. The precise meaning (as described by the transition rules) of the timed entry call will depend on the chosen notion of time. For example, let C1 be associated with the calling unit and C2 be associated with the called site. Note that this is not specified by the program, rather it is an instance of the distributed semantics. Consider point time for the following explanation. Let the map associated with C1 be as follows.

10	11
15	15
20	18
...	...

Let req_sent be generated at the calling site at time 10 with respect to c1 and req_resp be received at the calling site at time 20 with respect to c1. As the above specification requires req_resp to be measured with c2, the time to be used is actually 18 as returned by the map. If the deadline were 19, the first specification (with both events measured with respect to clock cl) will be violated while the second specification wil be satisfied. Notice that a message requesting time from C2 was not dispatched to know at what time req_resp occurred.

The following specification requires that an acknowledgement of the request be received within the deadline:

occur(req_sent,*,$) **wrt** cl **before** **occur**(time_recd,*,$$) **wrt** cl **atmost** D

This specification can also be modified to account for the discrepancies between clocks cl and c2. Certain plausible specifications on the called unit are now presented. Consider

occur(req_sent) **wrt** cl **before** **occur**(req_resp) **wrt** cl **atmost** D

It is identical to the first specification discussed but refers to the called site. Note that in the above specifications the temporal limit is equal to the maximum permissible delay. This can be altered depending on the network characteristics expected. So, if the average message transfer time is T units, the 'D' in the above specification can be changes to D-T. Of course, this is meaningful only if T is less than D. The specification described below, specifies a deadline (F) on the generation of 'time_recd' after a request has been received:

occur(req_sent,*,$) **wrt** cl **before** **occur**(time_recd,*,$$) **wrt** cl **atmost** F

If one wants to relate the time at which the request was sent by the calling site and the time at which the call was accepted at the called site, the value field of 'req_resp' is augmented to return the time the call was accepted. As timing errors depend on the value field a temporal specification cannot be defined. The event handler for 'req_resp' will check the value field and take the appropriate action. As the evens req_sent, req_resp and time_recd are generated on both the calling sites and the called sites, the specifications can be replicated on both the sites.

This concludes the first part of our discussion to demonstrate the expressiveness of ARL. In the next two sections we compare ARL with ESTEREL and real-time Prolog.

6.5 Comparison between ARL and ESTEREL

The purpose of this example from [10] is to compare the language ESTEREL and ARL. We describe the problem and present both the ESTEREL code and the ARL code. We conclude by justifying our claim that ARL is more expressive than ESTEREL. The problem for which solutions are presented is described below.

6.5.1 A reflex game

The game starts when a *reset* button is pressed and is composed of ten reflex measures. Each measure starts when the player presses the button *a*. After a random time a green lamp is lit after which the player must press a button *b* as fast as possible. The green lamp is then turned off and the reflex time displayed. A new measure starts when the player presses *a* again. When the cycle of ten measures is completed, the average reflex time is displayed after a pause of 3 seconds. During the process, it is possible to commit mistakes on which a bell sounds. It is also possible to cheat, upon which the game is reset. The possible list of mistakes/cheating together with the actions taken upon the occurrence is given below.

- *b* instead of *a* to start : Bell
- *a* during measure : Bell
- *b* before green lamp : Cheating: start a new game
- *a* or *b* not pressed within 10 seconds : Cancel current game and restart
- *reset* anytime : Start new game

6.5.2 The ESTEREL Program

The following is a verbatim reproduction of the program given in [10]. The general idea is that at any point in time one is expecting a particular event. If an unexpected event occurs, the error action is taken and the wait for the required event continues. When the event does occur, appropriate action is taken. There are deadlines attached to each of these waits. If the deadline is violated the game is restarted (as per specification.)

```
input pure signal RESET, A, B, MS
output pure signal GREEN_OFF, GREEN_ON, RED_OFF, RED_ON, RING_BELL
output single signal DISPLAY(int) IN
 every reset do
   emit RED_OFF;
   trapfailure
     var AVERAGE := 0 int in
     % measure loop
     repeat 10 times
```

```
      % waiting for A
      do
        do
          every B do emit RING_BELL
        upto A
      watching 10000ms abnormal failwith END_GAME end;
      % delay and waiting for B A rings the bell
      {
        everynext A do emit RING_BELL end
      ||
        % random delay -B may not be pressed
        do
          awaitnext random() MS
        watching B abnormal failwith END_GAME end;
        emit GREEN_ON
        % waiting for B and displaying result
        var time := 0 int in
          do
            do
              every MS do time := time + 1 end
            upto B
          watching 10000ms abnormal failwith END_GAME end
          emit GREEN_OFF
          emit DISPLAY(TIME)
          AVERAGE := AVERAGE + TIME
        end
      }
    end;
    %final display of the average time
    await 3000MS
    emit DISPLAY(AVERAGE/10)
  end
  failure END_GAME do emit RED_ON end
 end
end
```

6.5.3 The equivalent ARL program

A general overview of the ARL solution is described following which the ARL program is developed. Associated with each button and lamp is an event type. Events of the appropriate type are instantiated when the buttons are pressed or the lamp is switched on/off. In order to filter out erroneous events (like pressing B before A etc.), a module called 'filter' is introduced. It takes appropriate action for erroneous events. It communicates the events in the expected sequence to a module called 'main'. This is achieved by introducing two new events 'A_sig' and 'B_sig' which represent the correct A and B respectively. There are three temporal specifications dealing with the occurrence of A_sig, B_sig and the green light. Described below is the ARL program.

```
clock_pool c is
 (auto clock := clock + 1)
end c

type_pool signal_value is
values :: On | Off
end signal_value

with signal_value
event_pool signals is
{ A : button a, B : button b, reset_game : the reset button
Gl: green light, Bell: bell}
 event Gl :: values
 event A,B,Bell,reset_game ;
end signals

event_pool signif_signals is
{ These signals represent the actual signals after the spurious signals have been
filtered out.}
 event A_sig, B_sig
end signif_signals

with signals, signif_signals
module filter is
{ This module filters the unexpected events (like the occurrence of A when B is
expected and vice-versa) after issuing the appropriate error message. The correct
events are passed onto the module 'main'. }
A_occ : boolean { A_occ if true is to be interpreted as that only event A should
occur, occurrence of B is an error. }
a_handler(A) causes true
b_handler(B) causes true

  a_handler = A_occ := false; generate(A_sig), A_occ
            = generate(Bell),otherwise

  b_handler = A_occ := true; generate(b_sig), A_occ = false
            = generate(Bell),otherwise
```

```
{ Note the equational style of the above two functions. The body of the function
follows the '=' sign and the condition following the ','. But it uses the ':=' operator
and hence not a function.}

init
{ Initialization code for the module }
{ At first event A should occur}
A_occ := true;
end filter

{ The following is the main module}
with c, signals, signif_signals
module main is
 t: integer;
 sum_t: integer;
 indx : integer
 limit : integer constant := 10

{ The temporal requirements: default use of wrt field }
 label1 : occur(A_sig,*,$) before occur(B_sig,*,$$) atmost 10
 label2 : occur(B_sig,*,$) before occur(A_sig, *+1,$$) atmost 10
 label3 : occur(B_sig,*,$) after occur(Gl,On,$$) atmost 10

 procedure reset return boolean is
 begin
  t := 0;
  sum_t := 0;
  indx := 0;
  return true;
 end; || reset

{ Procedure start is invoked when an event of type A_sig has been received. It
switches on the green light after a random delay.}
 procedure start return boolean is
 begin
  delay random;
  t := clock.current_time;
  generate(Gl,On);
  return true;
 end; || start

{ Procedure stop is invoked when an event of type B_sig has been received}
 procedure stop is
 begin
```

```
{ Switch off the light}
   generate(Gl, Off)
   t := clock.current_time - t
   sum_t := sum_t + t
   indx := indx + 1
   if indx = limit then
{ If the required number of readings have been obtained, display the result and
prepare for next game}
      delay 3;
      display(sumt_t/limit);
      reset;
   end if;
   return true;
 end;|| stop

{ Event handler for temporal violation}
 tv_handler(temporal_violation) causes true
{ Event handler for reset_game }
 reset(reset_game) causes true

{ Handler definitions for events A_sig and B_sig }
 false(A_sig) causes start
 false(B_sig) causes stop

{ Equational with pattern matching definition of tv_handler }
 tv_handler (label1,order) = generate(Bell)
{ The above will never occur due to the filter but present for completeness }
 tv_handler (label1,time_out) = reset
 tv_handler (label2,order) = generate(Bell)
{ The above also will never occur due to the filter but present for completeness }
 tv_handler (label2,time_out) = reset
 tv_handler (label3,order) = reset
{ The above indicates cheating.}
 tv_handler (label3,time_out) = reset

end main
```

6.5.4 Analysis

Having written a program in two languages viz., ESTEREL and ARL, we analyze the two languages for expressiveness of constructs, programming style and flexibility. As it is not possible to demonstrate all the features of ARL in a program, our analysis consists of two parts. The first compares the constructs used in the program, while the second discusses features present in one but not the other.

The main point to observe from the programs is that in ARL, the specification of procedural control and the timing requirements are decoupled. Therefore it is possible to expand a program with more event handlers and temporal specifications without drastically affecting the existing program. In other words this decoupling leads to flexible programming, which is essential for prototyping systems.

For example, let there be an outer loop with a number of nested loops. Each of the nested loops has a time bound, while the outer loop also has a time bound. In ESTEREL this would be implemented with a number of *watching* statements. In ARL it would be implemented using events and the timing requirements would use these events. Any change in the timing requirements results in a change only to the temporal specifications without having to follow the control flow. In ESTEREL however, one has to know identify the temporal requirements with each loop. This is mainly because of the synchronous nature of events in ESTEREL due to which constructs like 'watching' etc., have to be used with the control flow.

However in ARL, due to the declarative style of event handling and the starting of a new thread on the occurrence of an event, a filter module is necessary to ignore erroneous events. This was not essential in ESTEREL as one could look out for events within a control loop (the 'every' statement.) But it is difficult to incorporate them in temporal specifications as they are not a part of the 'watching' statement.

Due to the de-coupling of the timing requirements and control flow in ARL, the construction of a temporal model is easier. Only the temporal statements have to be considered. Of course, determining whether they will be satisfied will be difficult in both the languages as satisfaction depends on the behavior of the program in question. In the ARL style of temporal specification, it is easier to check off that the list of timing requirements specified by the informal specification are provided by the program.

Certain other drawbacks such that the semantics of 'await' etc. were discussed in section 2.5 (chapter 4). Features present in ARL which have no corresponding feature in ESTEREL are polymorphisms, type inferencing, multiple clocks, quantified timing specifications.

6.6 A Lift System

This example, chosen from [30], concerns the specification of a lift system. The authors present the specification informally and develop a logic program for it. The system whose behavior is to be specified consists of a N lift system to be installed in a M floor building. The movement of lifts between floors is governed by the following constraints

- Each lift has a set of buttons one, for each floor. When pressed these cause the lift to visit the corresponding floor.
- Each floor has two buttons one to request an up-lift and one to request a down lift. The top and the first floor have only 1 button.
- When a lift has no requests to service it should remain at its final destination and await further requests.
- All requests for lifts from floors must be serviced eventually, with all floors given equal priority
- All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel

6.6.1 Logic Program

The specification as given by the authors is not a real-time system because there are no timing constraints that are to be satisfied. The specification however does involve time. For example, arrived(L,F,T) means that lift L arrived at floor F at time T. It does not require the lift to arrive at time T. The complete logic program along with a commentary is as follows.

departure(L,F,up,T_a)<- arrival(L,F,T_a), queue(L,Q,T,T_a), not (Q=[ ]), head(Q) > F.
If the queue is not empty, and floor F need not be serviced, the lift moves ahead. There is a similar rule for the lift going downwards. In this and and subsequent specifications, queue(L,Q,t1,t2) stands for the fact that a queue Q is associated with the lift L for the time period [t1, t2]

stop(L,F,T_a)<- arrival(L,F,T_a), queue(L,Q,T,T_a), head(Q) = F.
If floor F needs service, an arriving lift waits there.

stop(L,F,T_a)<- arrival(L,F,T_a), queue(L,[ ],T,T_a)
A lift halts at the floor where an empty queue is detected. This is because there could be more than one lift in motion and the last request is serviced by one of them. The rest of the lifts come to a halt.

departure(L,F,up,T_a+dts) <- stop(L,F,T_a), queue(L,Q,T,T_a+dts), head(Q) > F.
After stopping for a fixed amount of time(dts), a lift moves upwards if there is a floor greater than the current floor to be serviced. There is a similar rule for a downward motion.

departure(L,F,up,T_p)<- stop(L,F,T_a),queue(L,Q,T_p,T$-a$), $T_p > T_a + dt_s$,
queue(L,[ ],T_a+dt_s,T_p), head(Q)>F.
If no floors are to be serviced, the lift will halt at the current floor (where it has halted) only move when the queue becomes non-empty. It will leave in the direction of the request. Shown above is the rule for an upward motion. A similar rule will govern the downward motion.

arrival(L,F+1,T+dt)<-departure(L,F,up,T)
arrival(L,F-1,T+dt)<-departure(L,F,down,T)
These rules specify the time it takes to move up/down a floor after departure.

standing(L,F,T,T+dt_s)<-stop(L,F,T).
If a lift stops it waits atleast for dt_s.

standing(L,F,T_s,T)<-stop(L,F,T_s),queue(L,[ ].T_s+ dt_s,T)
A lift is standing as long as no floors are to be serviced.

moving(L,F,D,T,T + dt)<- departure(L,F,D,T).
On departure at time T, the lift enters the moving state after time T + dt.

standing(L,F,T_3,T_4) <- standing(L,F,T_1,T_2), $T_1 \leq T_3 < T_4 < T_2$.
moving(L,F,T_3,T_4) <- moving(L,F,T_1,T_2), $T_1 \leq T_3 < T_4 < T_2$.
If a state is proven to hold in time interval [T1,T2], it is possible to infer that it holds for all time intervals [T3,T4] included therein.

6.6.2 Discussion

The program while specifying the properties of the lift system, does not mention how the queues should be manipulated. The assumption that head(Q) has the relevant data is made. The authors justify this by claiming that manipulation of the queue is a scheduling policy. In our opinion, the scheduling policy was omitted because it is difficult to write a logic program to do so. Every rule developed above has to be augmented with calls to the routine which manipulate the queue. This could make the resulting program difficult to read. It also obscures the logic associated with the system, which is the principal goal of logic programming. In the next section we present the ARL program followed by a comparison between the two languages.

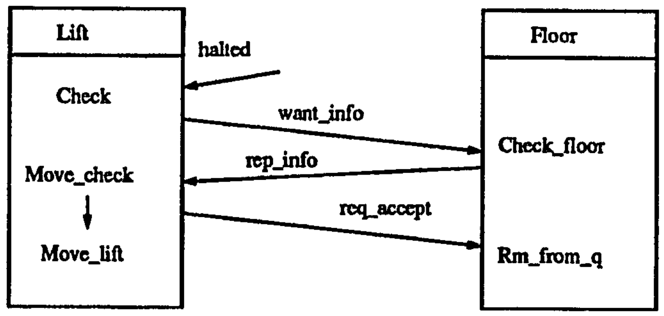


Figure 6.4. Possible Communications between a lift and the floor

6.6.3 The ARL program

The general structure of the ARL program is described below. Each lift is mapped onto a module. The floor system is mapped onto one module. The state of the lift is characterized by whether it moving or stationary, the floor it is at, the direction it is moving in (up/down) and the next floor it is to service. Each lift also has a list of floors to be serviced. This list (list_queue) is generated due to requests from the lift itself. When a lift comes to a halt, it generates an event called 'halted'. This causes it to check if it has any floors to service from an internal request. If so, its direction of further motion is fixed. Otherwise it can move in any direction. The lift then requests information from the floor database (called floor) regarding any service necessary by generating an event called 'want_info'. The floor database responds to the request by searching its data-base and replying to the original message via an event 'reply_info'. The lift selects the closest floor (either in the fixed direction or from current position) and services it. If the floor serviced was due to a request from the 'floor', a message (via an event called 'req_accept') indicating that the request was accepted is sent to the floor. The floor database assumes that a floor is not serviced until it has received a message indicating so. The communications between a lift and the floor data base is shown in figure 6.4.

The ARL program is described below.

```
type_pool comm_type is
  how_comm :: Any | Fixed
{ 'Any' indicates that the lift can change direction, while 'Fixed' indicates that the
lift must continue in its current direction }
end comm_type

with comm_type
```

```
event_pool communication is
   event want_info:: (integer,integer,how_comm)
{ want_info specifies the direction of current motion, the floor at which a lift has
halted and the direction the lift can move next }
   event rep_info, req_accept :: (integer,integer)
{ the first field represents the direction with 1 being up -1 being down and 0 pre-
senting halt, while the second field represents the floor number }
end communication

with communication,comm_type
module lift_handler[i: 1 .. MAXLIFT] is
{ A parametrized Module }
   input rep_info
   output want_info
{ The above defines a restriction on the use of events }
event halted :: (integer, integer) {local event}

state :: Moving | Stationary
{ Local enumerated type indicating whether the lift is in motion }
my_state : state;
my_floor : integer;
my_dir : integer;
next_floor : integer;
{ The above variables define the state of the lift }

{ Move the lift in direction dir and halt at floor number hlt }
procedure move_lift (dir,hlt:integer) return boolean is
begin
   my_state := Moving
   my_dir := dir
   while (my_floor /= hlt) loop
      my_floor := my_floor + dir
{ Moving to next floor takes time dt}
      delay dt
   end loop
{ Come to a halt and take appropriate action }
   my_state := Stationary
   generate(halted, (dir,hlt))
   delay wt { wait at the floor for some time }
return true
end

false(halted) causes check(halted)
{ Lift has halted and want to know where to move next }
```

```
procedure check(dir, hlt : integer) return boolean is
begin
   next_floor := lcl_check(dir,hlt)
   if next_floor = 0 then
      generate(want_info(dir,hlt,Any))
{ No floor to be serviced from the lift. Can move in any direction }
   else
      generate(want_info(dir,hlt,Fixed))
{ Has a floor to service and can move only in the fixed direction }
   end if
   return true { to prevent generation of disaster }
end

{ On reply from 'floor' check what to do next }
false(rep_info) causes move_check(rep_info)

{ Equational definition of checking where to move next when
request from floor has been received }
move_check (dir,hlt) = move_lift(dir,hlt); generate(req_accept(dir,hlt));
                       return true, dir /= my_dir
                       = move_lift(my_dir,next), my_dir = dir
                          where
                          next = best_max(next_floor,hlt), dir = -1
{ If moving down the highest requested floor is closest to the lift }
                          next = best_min(next_floor,hlt), dir = 1
{ If moving up the lowest requested floor is closest to the lift }


{ best_max selects the closest floor when moving downwards, i.e. the floor with the
maximum floor number. If the floor selected was due to the request from the floor
data base, the floor data base is informed about it.}

best_max x y = x; rm_lclq x, x > y
             = y; generate(req_accept(my_dir,y)) , y > x
             = y; rm_lclq x; generate(req_accept(my_dir,y)) , y = x

{ best_min selects the closest floor when moving upwards, i.e. the floor with the
minimum floor number. The floor data base is informed if necessary. }

best_min x y = x; rm_lclq x, x < y
             = y; generate(req_accept(my_dir,y)) , y < x
             = y; rm_lclq x; generate(req_accept(my_dir,y)) , y = x

{ Check if any locally generated floor is to be serviced }
```

```
lcl_check (dir,floor) = 0 , tg = [ ]
                      = max tg, dir = -1 & tg /= [ ]
                      = min tg, dir = 1 & tg /= [ ]
                        where
                        tg = [x|x<-lift_queue, (x > hlt & dir =1) or
                           (x < hlt & dir = -1)]

false(floor_but) causes add_lclq(floor_but)
{ floor_but is to be generated "externally" and represents floor selection from lift }

add_lclq x = lift_queue := lift_queue ++ [x]

rm_lclq x = lift_queue := lift_queue - [x]

{ add_lclq and rm_lclq have a shared object and their execution cannot be inter-
leaved }
excl { add_lcl_q, rm_lcl_q}

{ There is no initialization code as an external request from the floor will acti-
vate the lift }
end


{ The floor data base manager }
with Communication,Comm_type
module Floor_dbase is

   output rep_info
   input want_info, req_accept
{ Restriction of events }

{ A lift requests information }
false(want_info) causes check_flr_req(want_info)


{ check the queues for relevant information }
chk -1 x how = [ ], queue = [ ]
             = tg(1), tg /= [ ]
             = chk 1 x Fixed, how = Any || to avoid infinite recursion
             = [ ] , otherwise
               where
               tg = [(-1,z) | (-1,z) <- queue; z < x ]

chk 1 x how = [ ], queue = [ ]
            = tg(1), tg /= [ ]
            = chk -1 x Fixed, how = Any || to avoid infinite recursion
```

```
                = [ ] , otherwise
                  where
                  tg = [(-1,z) | (-1,z) <- queue; z < x ]

{ The following sub-program is invoked by the event want_info }
check_flr_req (dir,hlt,how) = reply(reply_info, res); return true
                              where
                              res = chk dir hlt how, chk dir hlt how /= [ ]
                                  = (0,hlt) , otherwise

{ floor_req is a request from the floor. Save the request and activate lift }
false(floor_req) causes add_to_q(floor_req)

add_to_q (dir,hlt) = queue := queue ++ [(dir,hlt)]
                     generate(rep_info,(dir,hlt))
{ the generate activates the lift if idle }

{ the request from floor was accepted. So remove from queue }
false(req_accept) causes rm_from_q(req_accept)

rm_from_q (dir,hlt) = queue := queue -- [(dir,hlt)]

{ As queue manipulation cannot proceed concurrently the following restriction is
essential }
excl {add_to_q, rm_from_q}
end
```

6.6.4 Analysis

Though the ARL program is greater in size than the logic program, most of the code is to manipulate the queues which the logic program does not deal with. If one was not concerned about the executability of the program, it would be possible to characterize state changes by events and have temporal constraints involving them. In that case there will be a one-to-one map between the ARL program the logic program. For example, arrival(L,F+1,T+dt)<-departure(L,F,up,T) can be translated into **occur**(arrival_L_D,*,$+1) **after** **occur**(departure_L_U,*,$) **atmost** dt. In the above translation, arrival_L_D is to be interpreted as arrival of lift L from down, and departure_L_U is to be interpreted as lift L departing upwards, while dt is the maximum time it should take.

We have not added any temporal constraint. In our scheme, it is possible to augment the program with timing constraints by using the events generated by the program. Once again we emphasize that it is not essential to follow the control of the program in order to add these constraints. This has been stated repeatedly because we feel that it enhances the expressiveness of the language for rapid prototyping. The interpretation of the time field in the logic program is not specified. They could be considered to be either upper or lower bounds. It is possible to extend the language in order to be able to specify *both* an upper bound and a lower bound. For example, it is possible to specify arrival(L,F+1,T1)<-departure(L,F,up,T0), T1-T0 < Upper_bound and also arrival(L,F+1,T2)<-departure(L,F,up,T0), T2-T1 > Lower_bound.

The logic program becomes more complex if one has to specify recovery actions for temporal violations. As ARL generates pre-defined events in the case of timing errors, it is easy to specify subprograms which handle recovery from timing errors. In the logic program however, one has to detect timing errors explicitly. For example, temporal violation regarding the arrival and departure of lifts can be specified as temporal_viol(late)<-arrival(L,F+1,T1), departure(L,F,up,T0),T1-T0 > Upper_bound. Here the interpretation requires the system to assign values to T0 and T1. In short, as timing errors are related to control and time is to be maintained by the system, temporal specification are difficult to handle in a logic program.

Also note that features related to typing, general use of events (fault tolerance specification), multiple clocks etc., are not discussed in the logic language.

6.7 Conclusion

We have shown how our language can be used in various situations by specifying various systems in it. In each of the examples we have a varied amount of functional components and a varied number of temporal specifications. Though we did not specify the mapping of clocks to modules, their addition is a trivial task. We have also compared ARL with two of the existing languages and explained how it has overcome their deficiencies. In the next chapter we summarize the achievements of this research and outline areas that require further research.

CHAPTER 7

CONCLUSIONS

We summarize the main points of the thesis and present the significant achievements. We also identify certain areas where more research is necessary towards satisfying our goal of having a good formal model for distributed real-time computation.

7.1 Achievements

In this thesis, we have defined a language for distributed real-time programming called ARL. The main emphasis in the design of the language has been to enhance the expressiveness over existing languages for distributed real-time systems. Therefore, we have borrowed ideas from functional a language (Miranda) and incorporated them with other features essential for distributed real-time programming to create a wide-spectrum language.

ARL supports the whole of Miranda and hence features like polymorphic types and functions, type inferencing, abstract data types, higher order functions, lazy evaluation, Z-F iterators and pattern matching. The type inferencing mechanism has been extended to support inferencing in the presence of heterogeneous types which is useful in exploratory programming [85]. Even though purely functional languages have a good theoretical foundation they cannot be used to specify real-time computation. So, ARL is not restricted to a functional language. Rather, it is an imperative language with a identifiable purely functional subset. The identification is achieved by categorizing a program into functions, observers and procedures using inference rules similar to the ones described in [31].

ARL supports the definition of explicit time in a program. The meaning of time is such that it is incremented independent of the flow of the program. In order to characterize distribution, ARL permits the definition of multiple clocks. The rationale being that if two different clocks are associated with two different syntactic units, the syntactic units can be assumed to be residing on different sites.

This view is not present in any language that we know of. This view is also used to distinguish concurrent systems from distributed systems. ARL by permitting mapping of modules to a clock allows the programmer to define a virtual node, using modules as the unit of distribution. But having multiple *completely independent* notions of time is not desirable. The various times are to be kept in synchrony with each other. The language provides constructs to specify which clocks are kept in synchrony and how often to synchronize. But ARL does not prescribe any algorithm to do so.

A distributed language must support a communication paradigm. We have chosen the asynchronous paradigm as it is inappropriate for a process in a real-time system to wait (when it could continue) for another process which has been delayed. Communication is achieved by instantiating an event. An event belongs to a particular type and has an occurrence number per module and a value associated with it. Events are also used in timing requirements. A declarative syntax is used to specify event handling. The semantics of the handler statement is powerful as to enable one to specify fault tolerance. However, we do not dictate any fault tolerance mechanisms.

Being a real-time language, ARL permits the specification of timing predicates. The timing predicates involve only the **after** and **before** operator. The predicates are based on event types and can depend on particular event values and occurrence numbers. They may also be universally quantified over the value and the occurrence fields. This idea can be considered to be an extension of polymorphic types and functions. The notion of events and universal quantification is also related to RTL [46] but the quantification in ARL is over both the value and the occurrence fields of the event type.

On the theoretical front, we have defined an operational semantics for the language based on dynamic algebras. We have demonstrated how resource restrictions, which play an important role in real-time computation, can be modeled. The semantics have been defined in a manner as to avoid any bias towards any implementation. This is achieved by parametrizing all implementation features such as nature of queues, scheduling policy etc. An implementation is free to choose the characteristics of such parameters to the semantics. The precise definition of these parameters will determine the goodness of the implementation. This abstraction does not sacrifice the operational nature of the semantics. It is our belief that an implementation of the language can be derived directly from the semantics.

In more detail, our key contribution has been in the semantic definition of a multi-clock system. We have shown how to model a system of clocks under clock synchronization in a functional way and also in a purely operational fashion. We have also discussed the representation of time when more than one clock is involved. The semantics maintains an interval representation of time. However, depending on the users needs, we have provided two options for representation of time in a program. The simple characterization of integer time is achieved by applying an averaging function to the interval, while an interval representation can be obtained by taking a sub-interval of the interval maintained by the semantics. We have also defined the meaning of distributed semantics by abstracting the mapping of clock(s) onto syntactic terms.

The meaning of the temporal constructs depends on the notion of time chosen. The use of integer time results in the standard definition of before and after, while the use of interval time one has to use interval temporal logic. Finally, we have shown how multiple clocks defined in a program can be used to define a distributed semantics by indexing the meaning of a program with the clock assignment function.

7.2 Future Directions

The development of a distributed real-time language and its semantics is only the first step to characterize distributed real-time computation. To really prove its usefulness in practice, the language has to be implemented. An implementation can either be directly based on the transition rules, or one could design an implementation by developing transformation rules to convert an ARL program into a readily available language like Ada. The second task appears to be easier, especially given our experience in building a distributed Ada system [95, 93, 56].

From a theoretical standpoint, one can study the relationship between the formal issues addressed by this thesis and other work to see how to derive models which have been used by others. This would simplify the task of building a model using the transition rules for ARL. In the following two sub-sections, some preliminary work in the above mentioned topics is discussed. First, we examine the translation of one of the ARL constructs to Ada. We also present preliminary definitions towards the generation of partial orders [74] from the transition rules for ARL.

7.2.1 ARL to Ada

Based on our experience in building distributed systems [95, 93, 56], it is our feeling that translating ARL programs to Ada programs is easier (from the view point of an implementor) than developing a full fledged ARL implementation. A reason for choosing Ada over other languages like C is that constructs for parallelism and time are available in Ada. This should ease the mapping from ARL to Ada. Of course, there is no one-to-one map between the features of ARL and Ada. For example, the notion of time is richer in ARL than in Ada. Therefore, one must used an extended definition of Ada (or atleast have a run-time system) to deal with multiple clocks and clock synchronization.

Note that one can always rewrite the transition rules in an Ada syntax, thereby achieving an ARL to Ada translator. However, this does not use all the Ada features and the translation is not very efficient. In the following paragraphs we illustrate how the Ada features of *generics* and *tasking* can be used effectively. We emphasize that more work needs to be done before the translation is complete and efficient.

7.2.1.1 Use of Generics

Polymorphic functions in ARL can be translated to generic functions in Ada. We explain how this translation can be achieved using an example. Recall, that a polymorphic higher order function until was defined as

```
until final trans state = state, final state
                        = until final trans (trans state),otherwise
```

The type of this function can be represented as $(* \rightarrow \text{boolean}) \rightarrow (* \rightarrow *) \rightarrow * \rightarrow *$. As there is one type variable, the generic will have one type parameter. Being a higher order function, the generic will also take functions as parameters. The equivalent Ada generic is

```
generic
    type star is private;
    with function final(paraml : star) return boolean;
    with function trans(paraml : star) return star;
    function until(state : star) return star;
end;

function until(state : star) return star is
begin
    if (final (state)) then
      return state;
    else
      return until(trans(state));
    end if;
end;
```

All instances of until as a concrete function (as opposed to a polymorphic function), results in a generic instantiation of the appropriate type. The above translation is valid only if until is instantiated with data types. But in Miranda, and hence ARL, 'state' can be a function. In such a situation, a different scheme is necessary. Such a scheme requires further research and is not explored in this thesis.

7.2.1.2 Use of Tasks

In this section, we show how the causal statement can be translated using tasks. Let $\{g_i(e_j)$: $i \in 1 .. n$, $j \in 1 .. m\}$ causes $f(e_{i1}, e_{i2}, \ldots, e_{ik})$ be a casual statement. For each of the g_i's there is an Ada task defined as

```
task body for_gi is
   lcl_v : val_type_i;
begin
   loop
      loop
         accept ei(v:val_type_i) do
                lcl_v := v;
         end;
         if (not gi(lcl_v) ) then
           exit;
         end if;
      end loop;
      rh_l.passive(i,to_glob(lcl_v));
      accept reactivate;
   end loop;
end;
```

The right hand side of the causal statement is handled by the following task.

```
task body rh_l is
      param : array ( 1 .. k) of glob_type;
begin
      loop
         for indx in 1 .. n loop
             select
                 ⋮
             or accept(i : integer;v: glob_type) do
                       if (i <= k) then
                         param(i) := v;
                       end if;
```

```
            end;
                    ⋮
          end select
      end loop;
      if (not f(to_type_1(param(1)), ..., to_type_k(param(k)))) then
        generate(disaster,name);
      end if;
      for_g_1.reactivate;
      for_g_2.reactivate;
      ⋮
      for_g_n.reactivate;
    end loop;
end;
```

If there is a limit on the number of active threads, the number of tasks can be reduced, with each task performing more functions. Note that some of the semantic parameters are implicitly defined by the above translation. For example, the rules regarding the **accept** statement force a FIFO handling of events, while the semantics left the handling of events as a scheduling decision.

7.2.2 Partial Orders

In this section, it is our aim to derive a relationship between the work done by others in modeling systems and our definitions. While there are a wide variety of models to choose from, we select the work of Pratt [74] in using partial orders to characterize concurrency as it is a fairly simple yet powerful model. Before we introduce our definitions, we describe Pratt's model briefly.

Pratt in [74] models concurrency (processes) using partially ordered multisets or *pomsets*. Rather than discuss the model in detail we present a precis. By process, an event oriented or state oriented computation is presumed in contrast with a functional style. A large number of operations on pomsets are defined, thus creating a pomset algebra. Just as strings model sequential computation, concurrency can be represented by partial orders. All the components that can occur concurrently have no temporal ordering imposed on them i.e., they are not comparable in the partial order. To be able to model realistic systems, the partial order has to be over a multiset as there could be repetitions in the computations. For example, a particular communication channel could be used repeatedly. A process can be looked upon as a collection of pomsets i.e., a set of pomsets. This is to be interpreted as a non-deterministic choice of one of the pomsets.

7.2.2.1 Partial Orders from Transition Rules

Introduced here are definitions which help build a partial order between events in a program from the semantics of ARL. Note that our characterization is based only on events. We do not consider mutable variables used by the program in the characterization. This is because events used in the a program signify important states. Towards that, let P be the given program. Define the set of configurations $\mathcal{B}$ (for behaviors) to be the function space from MODULES × EVENT_TYPES × INTEGER to VALUES × CLOCKS × INTEGER. That is it represents the function space of EVENT_OCCUR for each module. An element b of $\mathcal{B}$ of the form <(m,e,i) (v,c,t)> and is to be interpreted as the ith occurrence of event type e in module m occurs at time t when measured with respect to clock c and has value v. Let $\mathcal{M}_\sigma$ represent all possible behaviors of the program P when started in state σ. Clearly the elements of $\mathcal{M}_\sigma \subseteq \mathcal{B}$, i.e., $\mathcal{M}_\sigma \subseteq \mathcal{P}(\mathcal{B})$.

Intuitively it is clear that the notion of causality should determine the partial order. A formal definition of 'causality' is defined below. Let p,q,r be instances of subprogram invocations and let e,f,g be event types. An event is characterized by its type and occurrence number in a specified module. Let x, y denote events. For the purposes of the definition below assume that parameters to a subprogram can be represented by a single term.

Definition 7.1 A subprogram p *t_calls* (for transitive calls) another subprogram q with arguments 'param_1' if $[\![$ q(param_1) $]\!]$ is executed by p or if there exist a subprogram r and parameters 'param_2' and p executes $[\![$ r(param_2) $]\!]$ and r *t_calls* q.

Definition 7.2 Similarly a procedure p *generates* x, if p executes $[\![$ generate(x) $]\!]$ or if there is a q such that p *t_calls* q and q *generates* x.

Definition 7.3 An event x is said to *cause* a subprogram p, if the instantiation of x invokes p, or if there is a q such that x *causes* q and q *t_calls* p.

Definition 7.4 An event x is *causally before* an event y if there is a subprogram p such that x *causes* p and p *generates* y.

Definition 7.5 Define an ordering $\preceq$ on $\mathcal{B}$ such that <(m,e,i) (v,c,t)> $\preceq$ <(m',f,j) (w,c',t')> iff <e,i,t,c> is *causally before* <f,j,t',c'>.

Proposition 7.1 $\preceq$ is an irreflexive partial order

Proof 7.1 The irreflexivity and anti-symmetry are obvious. Assume towards a contradiction that there exists events x, y and z such that $x \preceq y \preceq z$ but not $x \preceq z$. As $x \preceq y$, there is a subprogram p such that x causes p and p generates y. And as $y \preceq z$, there is a subprogram q such that y causes q and q generates z. By the definition of cause, x causes q. Therefore $x \preceq z$, a contradiction.

In the above definition we have only shown how partial orders can be generated. We have not shown how the operators introduced in Pratt's model can be used. This requires further research. However, it is our belief that the definition of *generates* and *cause* can be extended to model the operators.

7.2.3 Replaceability

In this section we introduce a notion of *replaceability.* A notion of equivalences (of terms in the semantic language for ARL) will be useful in the development of a proof theory. Pragmatically, the idea of equivalences can be used to design fault-tolerant systems. For example, assume that processes P1 and P2 are equivalent. If due to a fault P1 cannot be executed, P2 can be used as a replacement for P1. The theory of equivalences has been studied in detail for concurrent systems [40]. However, the idea of equivalences is difficult to define in a real-time setting, as when an action is taken is important. We define a concept of *replaceability* which is similar to that of equivalence.

First, we assume that a real-time system can be characterized solely by events. Recall that an event is an instantiation of an event type and has a specific data value. Associated with the event is the time of occurrence as measured with a local clock. An event is therefore of the form < M,E,I,V,C,T>, where M is a module, E is the event type, I the instance number, V the value of the Ith instance, C the clock used to measure time of event occurrence and T the time with respect to C. Also recall the definition of a function called REMOTE_TIME_FUNCTION, which when presented with a time at a local clock returns the time at another local clock. For example, REMOTE_TIME_FUNCTION t c1 c2 = [t1,t2] is to be interpreted as that when local clock c1 shows time t, the local clock c2 could show a time anywhere between t1 and t2. Also note that a set of temporal predicates (involving events), which had to be satisfied could be derived from the program.

For the purposes of this thesis, assume that all relevant events are in the same module. Though all events occur in the same module, they can be measured with respect to different clocks. Hence, we drop the module part of the event and retain the clock field. Thus an event is represented by <E,I,V,C,T> where E,I,V,C and T are as before.

Definition 7.6 A pair $< e, i_e, v_e, c_e, t_e >$ and $< f, i_f, v_f, c_f, t_f >$ is said to be **relevant** to a temporal ordering formula **occur**(x, i_x, v_x) **wrt** c$_1$ $\oplus$ **occur**(y, i_y, v_y) **wrt** c$_2$ *limit* n iff
$[\ (x = e)\&(i_e = i_x \vee i_e \in i_x)\&(y = f)\&(i_f = i_y \vee i_f \in i_y)\&(v_e = v_x \vee v_e \in v_x)\&(v_f = v_y \vee v_f \in v_y)\ ] \vee$
$[\ (x = f)\&(i_f = i_x \vee i_f \in i_x)\&(y = e)\&(i_e = i_y \vee i_e \in i_y)\&(v_f = v_x \vee v_f \in v_x)\&(v_e = v_y \vee v_e \in v_y)\ ]$

In the above definition the 'element relation' is necessary to handle universally quantified formulae. Also note that the clocks associated with the formulae and the events need *not be identical.* Recall that the semantics used REMOTE_TIME_FUNCTION to relate the various clocks. We define satisfaction of a temporal formula for a pair of elements $< e, i, v, c, t >$ and $< f, j, w, c', t' >$. Define [t1,t2] to be REMOTE_TIME_FUNCTION(c, t, c_e) and [t3,t4] to be REMOTE_TIME_FUNCTION(c', t', c_f).

Definition 7.7 The pair $< e, i, v, c, t >$ and $< f, j, w, c', t' >$ is said to **satisfy** a temporal formula f of the form **occur**(e,oc$_1$,v$_1$) **wrt** c$_e$ **before occur**(f,oc$_2$,v$_2$) **wrt** c$_f$ **atmost** n iff (if they are **relevant then** (t2 < t3) and ((t4 - t1) $\leq$ n)).

The first term in the conjunction ensures 'before', while the second term requires that the maximum error in timing does not violate the condition. The definition of satisfaction for the **after** and **atleast** operator can be defined in a similar fashion. Note that the above definition is identical to the definition used in the semantics associated with interval time.

Definition 7.8 Let S be a set of events. S is said to **satisfy** a formula f iff: $\forall$ x,y $\in$ S: (x,y) **satisfy** f. Similarly, let F be a set (possibly infinite) of formulae. Then S **satisfy** F iff $\forall$f $\in$F S **satisfy** f.

Example 7.1 Let the system contain the event types E, F and G. Let (E,10,) **before** (F,,) **atmost** 5 be a temporal formula. Let S1 be the following set of events { <E,10,V,C,[1,2]>, <F,J,W,C,[4,5]> } and let S2 be the following { <E,11,V',C,[10,11]>, <G,K,U,C,[9,10]> }. By our definition of **satisfy**, both S1 and S2 satisfy the formula. S1 has relevant events which are within time bounds, while the events in S2 are not relevant.

Recall that **temporal_violations** was an event type such that events of its type indicate the occurrence of a temporal violation.

Definition 7.9 Define the **badness** of a set of events as the number of events of type **temporal_violation.** The **badness** of a set of such sets is the maximal number of the **badness** of each of the individual sets.

We define a notion of replaceability. It has two major components to it. The first component depends on a given set of input events and sets of output events. We allow sets of output events to handle non-deterministic behavior of program. In a non-deterministic program, it is possible to get various outputs given a single input.

Definition 7.10 For given non-empty sets $\mathcal{O}$ and $\mathcal{S}$, whose elements are sets of events, $\mathcal{S}$ is said to be **acceptable** with respect to $\mathcal{O}$ iff ($\mathcal{O} \subseteq \mathcal{S} \vee \mathcal{S} \subseteq \mathcal{O}$).

This definition is to ensure that the behavior of the system we are modeling is "not changed drastically" but can characterize non-determinism. The first part of the disjunction is to be interpreted as the system associated with $\mathcal{S}$ exhibiting more non-determinism than $\mathcal{O}$ while the second part is to be interpreted as the system exhibiting less non-determinism. This definition is similar to the one in [67].

Example 7.2 Let O = { {E,F}, {F,G}, {E,G} }. Let S1 = { {E,F}, {F,G} } and S2 = { {F,H} }. S1 is acceptable with respect to O, while S2 is not.

Definition 7.11 *Define the* **good_set** *of a set of events E, as the maximal subset of E containing no* **temporal_violations**.

The second component in the definition replaceability depends on the definition of satisfaction. For this assume, that states in a computation graph have information regarding events that have occurred. Also assume the temporal formulae that need to be satisfied by all states is defined. Let the set of formulae to be satisfied by the computation be $\mathcal{F}$.

Definition 7.12 A node N1, in a computation graph is **formula_replaceable** by another N2, iff (for every formula f in $\mathcal{F}$: if the events in N1 **satisfy** f then the events in N2 satisfy f).

In the above definition, events in N2 may satisfy more formulae than in N1. What is required is that replacement does not invalidate valid formulae. Therefore N1 and N2 may not be equivalent.

Definition 7.13 Let the computation graph obtained by replacing an event E in a given state $\mathcal{S}$ by another event F and re-executing the transition rules be called a **substitution graph.**

Definition 7.14 A graph Gl is said to be **replaceable** by Graph G2 iff (The **good_set** of events associated with the root node of G2 (R2) is **acceptable** with respect to the root node of Gl (Rl) and the **badness** of set of events of R2 is no more than the **badness** of the set of event of Rl and Rl is **formula_replaceable** by R2 and either for every successor of Rl there is a successor of R2 such that the sub-graph rooted at these successors are **replaceable** or for every successor of R2 there is a successor of Rl such that the condition holds.

As in the definition of **formula_replaceable,** the idea of **replaceable** is that the new node can take the place of the old node, if it is "no worse" than the old node. "No worse" has two components to it. The first is that the number of temporal violations are no more than before while the second is that all the original formulae are satisfied.

Definition 7.15 An event is said to be **replaceable** by another in a state S iff the original computation graph is **replaceable** by the **substitution graph.**

It is our opinion that these definitions form a basis for the development of a theory regarding real-time equivalence. However this requires further research and is out of the scope of this thesis.

Potential research in other areas can also be linked to our work. One other area is in the development of a 'theory of implementations' for real-time languages. For example, one can define a program to be *completely implementable* if for all "degrees of safety" there is an implementation, where a degree of safety can be considered to be the probability of a certain number of timing errors. The above can be expressed formally as $\forall$ P $\in$ L, $\forall\ \delta\ \in [0,1]\ \forall$ n $\in\ \omega\ \exists$ I $\in$ Implementation probability(I generates greater than n temporal violations) $\leq\ \delta$

The difficult part in the above formula is to formally define the set of all implementations and how to measure the probability of a program generating temporal violations.

Another important topic not discussed explicitly in the thesis is the concept of fairness. A scheduler is said to be fair, if processes that are enabled infinitely often are executed infinitely often. This concept is too general to be useful for real time systems. For example, in a particular execution sequence only a subset of possible events could be relevant while there could be processes which are enabled but generate irrelevant events. What is necessary is a definition of fairness which is function on the number of timing violations. This concept could be closely related to be above mentioned theory of implementation.

APPENDICES

APPENDIX A

BNF-like Grammar for ARL

type_def :: id **::** definition
definition :: tuple_def | constructor_def | sequence_def | abstype
tuple_def :: id **::** (id [, id])
tuple_def :: **type** id **is record** fields [fields] **end record**
fields :: id **:** id terminator
constructor_def :: id **::** constructor [| constructor]
constructor :: cap_id [id]
constructor_def :: **type** id (id : id) **is record** fields [fields] case_opt **end record**
case_opt :: **case** id **is** case_field [case_field] **end case**
case_field :: **when** id => fields [fields]
sequence_def :: id [id]
sequence_def :: **type** id **is array** (range)
zf_expr :: [expr iterator qualifier [qualifier]]
zf_expr :: [generator | qualifier [qualifier]]
iterator :: | | //
generator :: pattern <- expr [, expr]
qualifier :: expr | generator

abstype :: **abstype** id [poly] **with** signature
abstype :: constructor_def [law] local_declaration
poly :: string_of_stars
signature :: [absfunc] **implement** absdefin
absfunc :: id **::** type_indication
absdefin :: subprogram_decl
type_indication :: id | type_indication -> id
law :: pattern => expr [, expr]
local_declaration :: **where** [declaration]

clock_def :: **auto** id := id + expr **init** expr
clock_sync :: **auto** (id >> expr) id := id
broadcast_sync :: **auto** (id [, id] : expr)

periodic_task :: id_opt **auto** id >> expr subprogram_call
id_opt :: | id
delay_stat :: **delay** until_opt expr **wrt** clock_spec_opt
clock_spec_opt :: id | id [expr]
until_opt :: | **until**
event_type_decl :: **event** id [, id] | **event** id [, id] :: id
event_restrict :: **input** id [, id] | **output** id [, id]
gen_stat :: **generate** (id , expr)
reply_stat :: **reply** (id , expr)
causal_stat :: handler [, handler] **causes** subprogram_call
handler :: id (id)
temporal_spec :: id : predicate binary_opt
binary_opt :: limit | temp_oper predicate limit
temporal_oper :: **before** | **after**
limit :: **atmost** expr | **atleast** expr
predicate :: **occur** (id , oc_option , val_option) time_option
time_option :: | **wrt** clock_spec_opt oc_option :: expr | star_opt | dstar_opt
star_opt :: * | expr + * | * + expr
dstar_opt :: ** | expr + ** | ** + expr
val_option :: expr | dollar_opt | ddollar_opt
dollar_opt :: $ | expr operator $ | $ operator expr
ddollar_opt :: $$ | expr operator $$ | $$ operator expr

subprogram_decl :: subprogram_keyword_opt id parameter_spec **is** declarations **begin** body **end**
subprogram_keyword_opt :: | **function** | **efunction** | **observer** | **procedure** | **subprogram**
subprogram_decl :: s̄ubprogram_keyword_opt id eqn_parameter_spec = body local_declaration
local_declaration :: **where** [declaration]
body :: [statments terminator]
exclusive_def :: **excl** { id [, id] }
obj_decl :: id : id | id | id : definition
declaration :: [type_def | event_type_decl | subprogram_decl | obj_decl]

subprogram_call :: id (expr [, expr])
subprogram_call :: id [expr]

type_pl_decl :: import **type_pool** id is [type_def] **end** type_pool_opt
type_pool_opt :: | id

event_pl_decl :: import **event_pool** id **is** [event_type_decl] **end** event_pool_opt
event_pool_opt :: | id

clock_pl_decl :: **clock_pool** id **is** [clock_def] **end** clk_pool_opt
clock_pool_opt :: | id

import :: [**with** id [, id]]
module :: import module_spec module_body init_opt
init_opt :: endm_opt | **init** body endm_opt
endm_opt :: **end** | **end** id
module_spec :: **module** param_spec_opt **is**
param_spec_opt :: id | id mod_param
mod_param :: [mod_seq]
mod_seq :: id : expr .. expr [, mod_seq]
module_body :: distribution_map [declaration] [subprograms | clock_sync | causal_stat |, temporal_spec | periodic_task | event_restrict]
terminator :: NEWLINE | ;
distribution_map :: **for** module_map **use** clock_map
module_map :: id | id [id]
clock_map :: id | id [id]

APPENDIX B

Functions used by the Semantics

The following are the functions used in the semantics.

- ADD_TO_LIST
- ADD_TO_TIME_TABLE
- AVERAGE
- CAUSED : **causation** $\rightarrow$ **subprograms**
- CAUSER : **causation** $\rightarrow$ $\mathcal{P}$ **(subprograms)**
- CAU_EV : **causation** $\rightarrow$ $\mathcal{P}$ **(event_types)**
- CAU_ID
- CAU_LAB
- CLK_ASSIGN
- ESTIMATION
- EVENT_OCCUR: (POSITIVE $\rightarrow$ ((VALUES $\times$ CLOCKS $\times$ POSITIVE) $\cup$ $\{ \perp \}$))
- EVENT_SCOPE : **causation** $\times$ **event_types** $\rightarrow$ $\mathcal{P}$**(subprograms)**
- EVHAN_ID
- EVHAN_STATE
- FIRST_CLK : **temporal_specification** $\rightarrow$ **clocks**
- FIRST_EV : **temporal_specification** $\rightarrow$ **event_types**
- FIRST_OC : **temporal_specification** $\rightarrow$ INTEGER $\cup$ { *, ** }
- FIRST_VAL : **temporal_specification** $\rightarrow$ **values** $\cup$ { $, $$ }
- GEN_NAME : **generate** $\rightarrow$ **event_types**
- GEN_VALUE : **generate** $\rightarrow$ **values**
- GET_EVENT_ENTRY
- INCR : **clocks** $\rightarrow$ INTEGER

- INITIAL : **clocks** → INTEGER
- INTERP_FROM
- INTERP_TO
- LIMIT : **temporal_specification** → { UPPER, LOWER }
- LIMIT_VAL : **temporal_specification** → INTEGER
- MY_MODULE : NODES → MODULES
- MY_SELF
- NEXT
- PARAMS
- PARENT
- PT_CLK : **periodic_tasks** → **clocks**
- PT_INTERVAL : **periodic_tasks** → INTEGERS
- PT_NAME : **periodic_tasks** → **subprogram**
- PT_PARAM : **periodic_tasks** → $\text{INTEGER}_{\textbf{(expr)}}$
- RECEIVE_MESSAGE
- REMOTE_TIME_FUNCTION: CLOCKS × INTEGER × CLOCKS → INTEGER
- REMOVE_FROM_LIST
- RESET
- REST_CLK
- ROOT
- SECOND_EV : **temporal_specification** → **event_types**
- SECOND_CLK : **temporal_specification** → **clocks**
- SECOND_OC : **temporal_specification** → INTEGER∪ { *, ** }
- SECOND_VAL : **temporal_specification** → **values** ∪ { \$, \$\$ }
- SEND_MESSAGE
- SET
- SPGM_PARAM : **subprograms** → $\text{INTEGER}_{\textbf{(expr)}}$
- SPGM_TYPE : **subprograms** → { Procedures, Observers, Functions, Efunctions }
- SYNCED : **clk_sync** → **clocks**
- SYNCER : **clk_sync** → **clocks**
- SYNCINT : **clk_sync** → INTEGERS
- SYNC_FUNC

- SUBPR_SCOPE: SUBPROGRAMS $\rightarrow \mathcal{P}$(EVENT_TYPES)
- SUFF
- TERM_TYPE
- TS_LAB : **temporal_specification** $\rightarrow$ **temporal_labels**
- TSPEC_CLK
- VISIBLE: MODULES $\times$ EVENTS $\rightarrow$ BOOLEAN.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] *Ada programming language (ANSI/MIL-STD-1815A)*, Washington, D.C. 20301, January 1983. AJPO : DOD, OUSD(R&D).

[2] G. A. Agha. *Actors: A model of concurrent computation in Distributed Systems.* The MIT Press, 1986.

[3] A. Aho, R. Sethi, and J. Ullman. *Principles of Compiler Construction.* Addison Wesley, 1987.

[4] J. F. Allen. Interval Logic. *Communications of The ACM*, 26(11), NOVEMBER 1983.

[5] P. America, J. de Bakker, J. N. Kok, and J. Rutten. Operational semantics of a parallel object-oriented language. *ACM Symposium on Principles of Programming Language*, 1986.

[6] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20-7:519–526, July 1977.

[7] J. Backus. Is Computer Science Based on the Wrong Concept of Programs. In J. DeBakker, editor, *Algorithmic Languages.* North Holland, 1981.

[8] J. Backus. Function Level Computing. *IEEE Spectrum*, August 1982.

[9] J. L. Bergerand, P. Caspi, D. Pilaud, N. Halbwachs, and E. Pilaud. Outline of a real-time data flow language. In *IEEE Real-Time Systems Symposium*, pages 33–42, 1985.

[10] G. Berry and L. Cosserat. The ESTEREL synchronous programming language. In *Seminar on Concurrency.* Springer Verlag LNCS-197, 1984.

[11] G. Berry, S. Moisan, and J. P. Rigault. ESTEREL: Towards a synchronous and semantically sound high level language for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 30–37, 1983.

[12] G. Booch. *Software Engineering with Ada.* Benjamin/Cummings, 1987.

[13] M. Broy. A fixed point approach to applicative multiprogramming. In Broy and Schmidt, editors, *Theoretical Foundations of Programming Methods.* Reidel Publishing Company, 1982.

[14] M. Broy. Applicative real time programming. *Proceedings of IFIP Congress, Paris*, pages 259–264, 1983.

[15] L. Cardelli and P. Wegner. On understanding types data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[16] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming systems. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 178–188, 1987.

[17] K. M. Chandy and J. Misra. Parallelsim and programming: A perspective. In *Foundations of Software Technology and Theoretical Computer Science, LNCS 287*, pages 173–194. Springer Verlag, 1987.

[18] K. L. Clark and S Gregory. Parlog :a parallel logic programming language. Technical Report 5, Imperial College, May 1983.

[19] L. Damas and R. Milner. Principle type schemes for functional programs. *ACM Symposium on Principles of Programming Language*, pages 207–212, 1982.

[20] R. B. Dannenberg. Arctic: A functional language for real-time control. In *ACM Symposium on Lisp and Functional Languages*, pages 96–103, 1984.

[21] A. Dapra, S. Gatti, S. Crespi-Reghizzi, F. Maderna, D. Belcredi, Natali, R. A. Stammers, and M. D. Tedd. *Using Ada and APSE to support distributed multimicroprocessor targets.* Commission of the European Communities, March 1982.

[22] J. W. deBakker, J. N. Kok, J. J. Ch. Meyer, E. R. Olderog, and J. I. Zucker. Contrasting themes in the semantics of imperative concurrency. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Current Trends in Concurrency : LNCS 224*, pages 51–121. Springer Verlag, 1986.

[23] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(5):569, September 1965.

[24] A. Diller. *Compiling Functional Languages.* John Wiley and Sons, 1985.

[25] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: On branching time versusn linear time temporal logics. *Journal of the Association of the Computing Machinery*, 37(1):151–178, January 1986.

[26] H. B. Enderton. *Elements of Set theory.* Academic Press College Division, 1977.

[27] M. Falaschi, G. Levi, and C. Palamidessi. A synchronization logic: Axiomatic and formal semantics of generalized Horn clauses. *Information and Control*, pages 36–69, 1984.

[28] A. A. Faustini and E. B. Lewis. A declarative language for the specification of real time systems. In *IEEE Real Time Systems Symposium*, pages 43–51, 1985.

[29] Y. C. Fuh and P. Mishra. Type inference with subtypes. In H. Ganzinger, editor, *2nd European Symposium on Programming, LNCS 300*, pages 94–114. Springer Verlag, March 1988.

[30] F. Garzotto, C. Ghezzi, D. Mandrioli, and A. Morzenti. On the specification of real-time systems using logic programming. In *ESEC-87, LNCS 289*, pages 180–190. Springer Verlag, 1987.

[31] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. *ACM Symposium on Functional programming and Lisp*, pages 28–38, 1986.

[32] J. A. Goguen, J. P. Jouannaud, and J. Meseguer. Operational semantics for order sorted algebra. In *ICALP -85, LNCS 194*, pages 221–231. Springer Verlag, 1985.

[33] A. Goldberg. *Smalltalk 80: The langauge and its Implementation.* Addison-Wesley, 1983.

[34] J. B. Goodenough and L. Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority ada tasks. *Ada Letters*, 8(7), Fall 1988.

[35] M. J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer Verlag, 1979.

[36] Y. Gurevich. Logic and the challenge of computer science. In E. Borger, editor, *Current Trends in Theoretical Computer Science.* Computer Science Press, 1987.

[37] Y. Gurevich and L. Moss. The algebraic operational semantics of Occam. Private Communication, 1988.

[38] J. V. Guttag. *The Specification and application to programming of abstract data types.* PhD thesis, Department of Computer Science, Univerity of Toronto, 1975.

[39] B. T. Hailpern. Verifying concurrent programs in temporal logic. Technical report, Department of Computer Science, Stanford University, August 1980.

[40] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association of the Computing Machinery*, 32(1):137–161, January 1985.

[41] M. C. B. Hennessy and W. Li. Translating a subset of ada into ccs. In D. Bjoerner, editor, *Proceeding of the IFIP TC2 Working Conference on Formal Description of Programming Concepts II*, pages 227–249. North Holland, 1982.

[42] C. Hewitt and H. Baker. Laws for communicating parallel processes. *IFIP -77*, pages 987–992, 1977.

[43] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[44] C. A. R. Hoare. *Communication Sequential Processes.* Prentice Hall International, 1985.

[45] C. Huizing, R. Gerth, and W. P. de Roever. Full abstraction of a real-time denotational semantics for an OCCAM-like language. *ACM Symposium on Principles of Programming Languages*, 1987.

[46] F. Jahanian and A. K. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, pages 961–975, August 1987.

[47] W. H. Jessop. Ada packages and distributed systems. Unpublished.

[48] J. E. Coolahan jr and N. Roussopoulos. Timing requirements for time-driven systems using augmented petri nets. *IEEE Transactions on Software Engineering*, pages 603–616, September 1983.

[49] B. W. Kernighan and D. M. Ritchie. *The C programming language.* Prentice Hall Englewood Cliffs, 1978.

[50] E. Klingerman and A. D. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949, September 1986.

[51] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, August 1987.

[52] R. Kowalski. Algorithm = Logic + Control. *CACM*, August 1979.

[53] R. Kowalski. The relation between logic programming and logic specification. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages.* Prentice Hall, 1984.

[54] P. Koymans, J. Vytopil, and W. P. deRoever. Real-time programming and asynchronous message passing. *ACM Symposium on Principles of Distributed Computing*, pages 187–197, August 1983.

[55] R. Koymans, R. K. Shyamsunder, W. P. deRoever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. In *Logics of Programs.* Springer Verlag LNCS-193, 1985.

[56] P. Krishnan, R. Volz, and R. Theriault. Implementation of task types in distributed Ada. *2nd Int'l Workshop on Real Time Ada Issues*, June 1988.

[57] F. Kroger. *Temporal Logic of Programs.* Springer Verlag, 1987.

[58] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[59] P. J. Landin. A λ-calculus approach. In L. Fox, editor, *Advances in Programming and non-numerical computation.* Pegamon Press, 1966.

[60] A. L. Lansky. *Specification and Analysis of Concurrency.* PhD thesis, Department of Computer Science, Stanford University, 1983.

[61] P. E. Lauer. A simple railway system. In *The Analysis of Concurrent Systems, LNCS :207.* Springer Verlag, 1983.

[62] G. Levi. Logic programming: The foundation the approach and the role of concurrency. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Current Trends in Concurrency : LNCS 224*, pages 396–441. Springer Verlag, 1986.

[63] G. J. Milne. CIRCAL and the representation of communication, concurrency and time. *ACM Transactions on Programming Language and Systems*, 7(2):270–298, April 1985.

[64] R. Milner. *A Calculus of Communicating Systems.* Lecture Notes on Computer Science Vol. 92. Springer Verlag, 1980.

[65] J. M. Morris. *Algebraic Operational Semantics of Modula-2.* PhD thesis, Computer and Communication Sciences, University of Michigan, 1988.

[66] P. Mosses. Abstract semantics algebras. In D. Bjoerner, editor, *Proceeding of the IFIP TC2 Working Conference on Formal Description of Programming Concepts II*, pages 63–88. North Holland, 1982.

[67] E. R. Olderog and C. A. R. Hoare. Specification-oriented semantics for communicating processes. In *ICALP -83, LNCS 154*. Springer Verlag, 1983.

[68] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Language and Systems*, 4(3):455–495, July 1982.

[69] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the Association for Computing Machinery*, 4(3), October 1979.

[70] J. L. Peterson and A. Silberschatz. *Operating System Concepts.* Addison Wesley, 1983.

[71] U. F. Pleban and P. Lee. High level semantics: An integrated approach to programming language semantics and the specification of implementations. In *Mathematical Foundations of Programming Language Semantics: LNCS 298*, pages 550–571. Springer Verlag, 1987.

[72] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[73] G. D. Plotkin. An operational semantics for CSP. In D. Bjoerner, editor, *Proceeding of the IFIP TC2 Working Conference on Formal Description of Programming Concepts II*, pages 199–225. North Holland, 1982.

[74] V. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1), 1986.

[75] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *ICALP -86, LNCS 226*. Springer Verlag, 1986.

[76] G. M. Reed and A. W. Roscoe. Metric spaces as models for real-time concurrency. In *Mathematical Foundations of Programming Language Semantics: LNCS 298*, pages 331–343. Springer Verlag, 1987.

[77] A. Rueveni. *The Event Based Language and its multiple processor implementations*. PhD thesis, M.I.T Laboratory for Computer Science, 1980.

[78] D. S. Scott. Domains for denotational semantics. In *ICALP -82, LNCS 140*. Springer Verlag, 1982.

[79] E. Y. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, ICOT, 1983.

[80] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the Association of the Computing Machinery*, 34(3):626–645, July 1987.

[81] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, pages 10–19, October 1988.

[82] J. A. Stankovic, W. A. Halang, and M. Tokoro, editors. *The International Journal of Time-Critical Computer Systems*, volume 1. Kluwer Academic Publishers, June 1989.

[83] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[84] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[85] S. Thatte. Type inference with partial types. In *ICALP -88, LNCS 317*, pages 615–629. Springer Verlag, 1988.

[86] D. Turner. The semantic elegance of applicative languages. In *ACM Symposium on Functional Programs and Computer Architecture*, 1981.

[87] D. Turner. Functional programs as executable specifications. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*. Prentice Hall, 1984.

[88] D. Turner. An overview of Miranda. *ACM Sigplan Notices*, December 1986.

[89] D. Turner. Functional programming and communicating processes. In J. W. deBakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE-II , LNCS 259*. Springer Verlag, 1987.

[90] D. A. Turner. The SASL language manual. Technical Report CS/75/1, Department of Computational Science, University of StÅndrews, 1976.

[91] D. A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, September 1979.

[92] R. Volz. Virtual nodes and units of distribution for distributed Ada. *3rd Int'l Workshop on Real Time Ada Issues*, June 1989.

[93] R. A. Volz, P. Krishnan, and R. J. Theriault. An approach to distributed execution of ada programs. *IEEE International Symposium on Intelligent Control*, January 1987.

[94] R. A. Volz and T. N. Mudge. Timing issues in distributed execution of ada programs. *IEEE Transaction on Computers*, April 1987.

[95] R. A. Volz, T. N. Mudge, G. D. Buzzard, and P. Krishnan. Translation and execution of distributed Ada programs: Is it Ada? *IEEE Transactions on Software Engineering*, pages 281–292, March 1989.

[96] R. A. Volz, T. N. Mudge, A. W. Naylor, and J. H. Meyer. Some problems in distributing real-time ada programs across machines. *Ada in Use, Proceedings of the 1985 International Ada Conference*, pages 72–84, May 1985.

[97] R. A. Volz and R. E. Richardson. Crash users manual. Technical report, Dept. of ECE University of Michigan, August 1977.

[98] D. A. Watt. An action semantics of standard ML. In *Mathematical Foundations of Programming Language Semantics: LNCS 298*, pages 572–598. Springer Verlag, 1987.

[99] P. Wegner. The vienna definition language. *ACM Computing Surveys*, 4(1), 1972.

[100] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, and M. Broy. On hierarchies of abstract data types. *Acta Informatica*, 20:1–33, 1983.

[101] P. Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, SE-8:250–269, May 1982.