# A VISUAL FRAMEWORK FOR FORMAL SYSTEMS DEVELOPMENT USING INTERVAL TEMPORAL LOGIC

**Ph.D. Thesis**

**De Montfort University**
**Software Technology Research Laboratory**

**Arun Chakrapani Rao**

**2002**

## Abstract

This thesis will give an introduction to *specification methods* for *real-time safety-critical systems* including *formal methods*. While formal methods offer various benefits for developing systems and software by virtue of their precise semantics, their uptake by a wider spectrum of users, including system and software engineers, is hampered by various drawbacks. The mathematical notations of formalisms form the main stumbling block in their comprehension and hence lead to associated accessibility problems. Visual languages are excellent candidates as a means to overcome this problem. However, most visual languages lack a well-defined formal semantics. Hence, the creation of a visual development suite supporting refinement and abstraction based on a well-defined formalism has been attempted. The *Interval Temporal Logic (ITL)* formalism is described in detail as it forms the basis for our development method. A study was conducted to see how visualisation helps in various domains in fostering increased accessibility of information, language and technology. Identifying a design rationale, a simple, intuitive and readable visual language, called *VisITL* with a well-defined formal semantics was designed. A supporting visual framework of refinement and abstraction rules was also devised. Examples are given depicting the application of these rules to VisITL specifications. Case studies undertaken to illustrate the use of this visual framework are described. The VisITL tool demonstrates the realisability of this approach. Comparisons to related work are made and suggestions are given for future efforts in this area.

## Abstract

This thesis will give an introduction to *specification methods* for *real-time safety-critical systems* including *formal methods*. While formal methods offer various benefits for developing systems and software by virtue of their precise semantics, their uptake by a wider spectrum of users, including system and software engineers, is hampered by various drawbacks. The mathematical notations of formalisms form the main stumbling block in their comprehension and hence lead to associated accessibility problems. Visual languages are excellent candidates as a means to overcome this problem. However, most visual languages lack a well-defined formal semantics. Hence, the creation of a visual development suite supporting refinement and abstraction based on a well-defined formalism has been attempted. The *Interval Temporal Logic (ITL)* formalism is described in detail as it forms the basis for our development method. A study was conducted to see how visualisation helps in various domains in fostering increased accessibility of information, language and technology. Identifying a design rationale, a simple, intuitive and readable visual language, called *VisITL* with a well-defined formal semantics was designed. A supporting visual framework of refinement and abstraction rules was also devised. Examples are given depicting the application of these rules to VisITL specifications. Case studies undertaken to illustrate the use of this visual framework are described. The VisITL tool demonstrates the realisability of this approach. Comparisons to related work are made and suggestions are given for future efforts in this area.

# Declaration

I declare that the work described in my thesis is original work undertaken by me between September 1997 to September 2000, for the degree of Doctor of Philosophy, at De Montfort University, United Kingdom.

This thesis is written by me and produced using LaTeX.

# Acknowledgements

I wish to sincerely acknowledge the help, motivation and encouragement provided to me by all my supervisors. In particular, I wish to thank Dr. Antonio Cau, my first supervisor, with whom I had regular contact and interesting discussions and Prof. Hussein Zedan, my second supervisor, with whom I had interesting discussions apart from a lot of encouragement and support. I also wish to acknowledge the help from my second "second supervisor", Dr. Ben Moszkowski, with whom I had several useful discussions and e-mail exchanges.

The source of inspiration and help that came from my colleagues at the Software Technology Research Laboratory is gratefully acknowledged.

The Research Methods Training Programme is another source which gave me the insight and courage to persevere with my research in a positive frame of mind. Thanks are due to the research unit for organising this and also helping me keep track of my own progress through assessment at regular intervals.

I also acknowledge the help received from staff and students at the EMTERC during my initial difficult months in research which also coincided with my first stay abroad.

Last but not the least, I wish to acknowledge the support I received from my wife Roopa, our family abroad; especially my brother, Praveen, and sister, Brinda. Their patience and encouragement has been invaluable.

I dedicate this thesis to my parents,

R. Chakrapani Rao and Prema Chakrapani.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Overview

In the beginning of the year 2002, more than 75 different formal methods were listed on the formal methods repository at the WWW virtual library on formal methods [53]. These formal methods are in different stages of development. While many have little or no tool support, some have good tool support and industrial users. Formal methods are evolving at a rapid pace and making inroads into industry. A major issue in technology transfer to industry from research laboratories involves tool support. The tool support not only has to be adequate but also appropriate. In other words, the tools have to support a wide variety of users. They should cater not only for undergraduate users in education but also to users in real projects. While formal methods continue to grow in popularity, a number of misconceptions regarding formal methods continue to grow in tandem [20]. The tools built should collectively overcome these misconceptions. Lean Formal Methods as mentioned for the first time in [26] suggest a way forward.

## Scope

The work broadly aims to contribute towards making formal methods more acceptable and convenient to a wide spectrum of users. It aims to integrate formal specifications into the system development process without burdening the users with many formal notations and the need to work with them manually. Example specifications are developed to show how this can be achieved in practice. Implementation challenges to be tackled are identified and future work directions are suggested.

## Original Contribution

In brief, there are various existing software development process models according to which software can be developed. These may involve formal methods and/or systematic methods to varying extents. So far, most methods incorporate only systematic methods and therefore lack formal mathematical underpinnings. The development of real-time safety-critical systems necessitates more formal approaches. Our research work involves a lean, formal approach where the accessibility of a formal method is tackled through visualisation approaches.

A formal method like Interval Temporal Logic (ITL) uses a lot of symbols. A user who has a lot of interest in Formal Methods (FMs) will not mind this and gets used to this pretty fast. However, users involved in other software development activities will not find a formal specification easy to read or understand let alone develop their own specifications. As a result, a simple graphical notation (called VisITL) was invented to enable the creation of more readable specifications. The design rationale for such a visual notation is highlighted. A development method involving visual refinement as well as abstraction is proposed. Case studies are given to illustrate the use of VisITL in a visual development framework.

## Thesis Outline

Chapter 2 explores process models and strategies in software development and introduces various formal approaches for the development of real-time, safety-critical systems. It also explains the rationale for the choice of Interval Temporal Logic (ITL) as the formalism for our work.

Chapter 3 provides an insight into how visualisation helps in various domains in fostering increased accessibility of information, languages and technology through various means including better communication and ease of use.

Chapter 4 first introduces Interval Temporal Logic (ITL) in detail and highlights some of the difficulties that have to be tackled to increase its uptake by a wider spectrum of users. Then, it details possible approaches to visualisation for ITL and introduces a visual notation for ITL called VisITL. The design rationale for such a visual notation is explained. Example specifications are given in the visual notation.

Chapter 5 introduces a development technique based on ITL and integrates it with VisITL by providing a visual framework for refining and abstracting VisITL specifications.

Chapter 6 involves case studies demonstrating the use of the visual framework.

Chapter 7 gives details of the VisITL tool.

Chapter 8 gives conclusions outlining further work in this area.

# Chapter 2

# Formal Software Development Techniques

The objectives of this chapter are to explore process models and strategies in software development, introduce formal approaches for the development of critical systems and state the rationale for the choice of *Interval Temporal Logic (ITL)* as the basis for my work.

## 2.1 Software Development - Process Models and Strategies

### 2.1.1 Software Development Process Model

The goal of any development effort is to produce a product. A development process is a set of activities, together with an ordering relationship between activities, which, if performed in a manner that satisfies this ordering, will produce the desired product. A process model is an abstract representation of a development process. The goal in a software development effort is to produce high "quality" software. The software development process is, therefore, the sequence of activities that will produce high quality software. The basic activities in a software development process include requirements analysis, specification, design, coding and testing which are further broken down into distinct activities.

The following discusses some of the process models [129] :

- **Waterfall model**

  The oldest and widely used process model is the *waterfall model* which states that the phases are organised in a linear order. A simple form of the waterfall model is described in Figure 2.1.

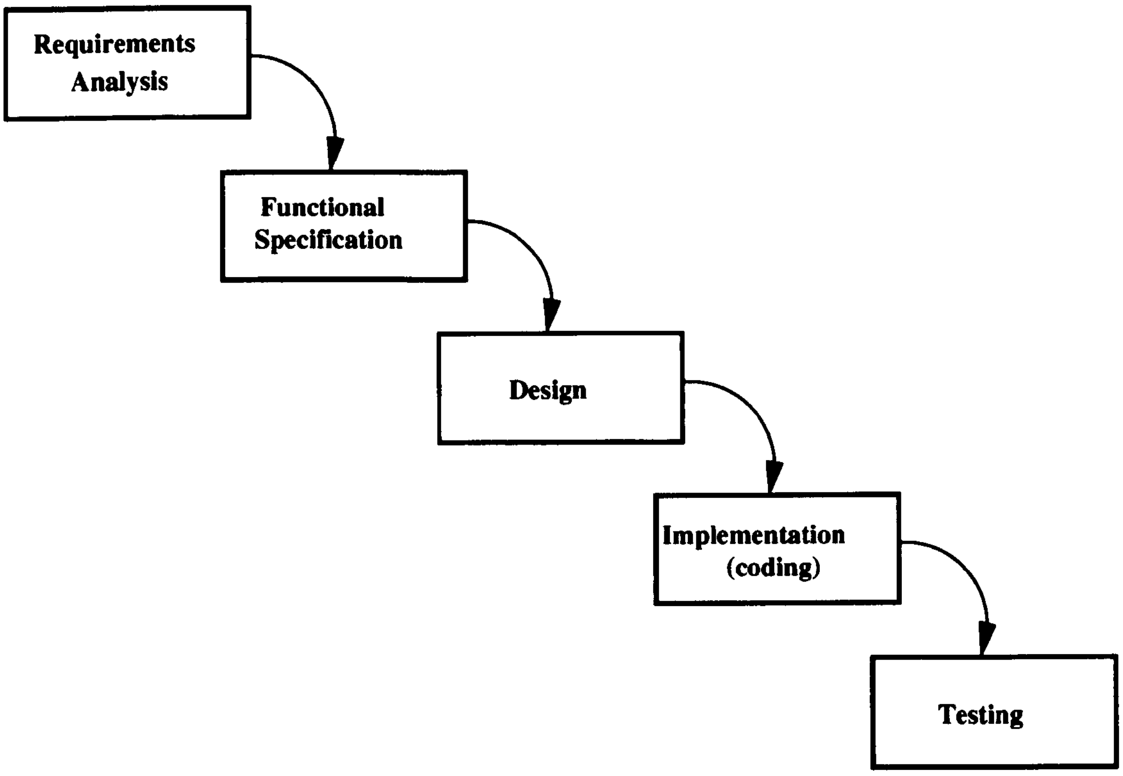


Figure 2.1: A Simple Form of the Waterfall Model

  A limitation of the waterfall model is that it assumes that the requirements of the system can be frozen before the rest of the process begins.

- **Prototyping based model**

  In this model, instead of freezing the requirements before the other phases, a throwaway prototype is built to help understand the requirements. The client can get an "actual feel" of the system since the interactions with the prototype can enable the client to better understand the requirements of the desired system.

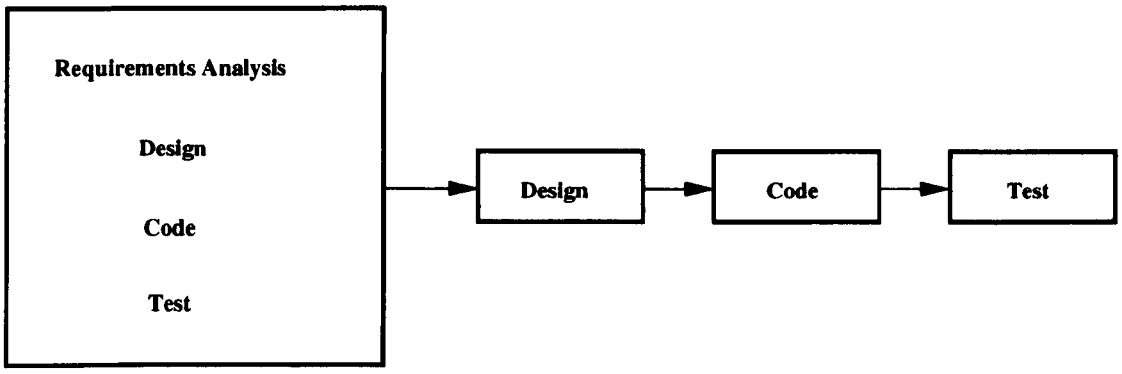


Figure 2.2: The Prototyping Model

The prototyping model is illustrated in Figure 2.2.

The cost involved in this build-it-twice approach is usually a major disadvantage.

- **Iterative enhancement model**

  The iterative enhancement model tries to combine the benefits of both the prototyping and the waterfall model. The basic idea is that the software should be developed in increments, each increment adding some functional capability to the system. An advantage of this approach is that it can result in better testing as testing each increment is likely to be easier. Also, the increments provide feedback to the client and hence, it helps in determining the final requirements of the system.

  The iterative enhancement model is illustrated in Figure 2.3.

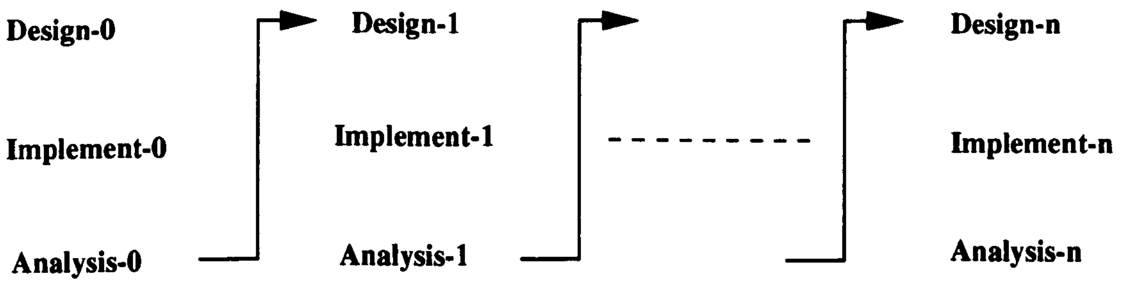


Figure 2.3: The Iterative Enhancement Model

- **Spiral model**

This is a model proposed by Boehm [17] where the activities are organised like a spiral. The spiral has many cycles. The radial dimension represents the cumulative cost incurred in accomplishing the steps done so far, and the angular dimension represents the progress made in completing each cycle of the spiral. The model is shown in Figure 2.4.

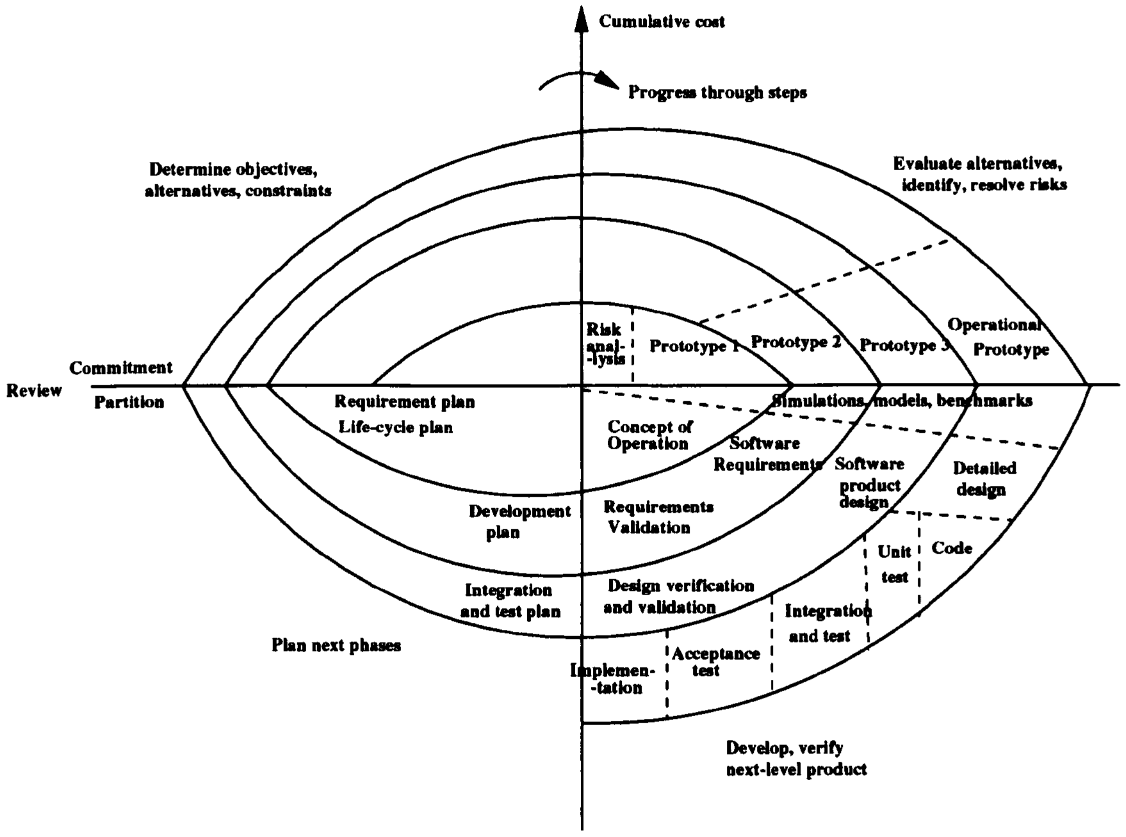


Figure 2.4: The Spiral Model

- **The transform model**

This is another model that Boehm has identified as being significant. It is based upon the automatic conversion of a software specification into a program that satisfies the specification. Little practical progress has been made in this area. Limited versions of this model offer more scope for real progress [54].

- **V-diagram**

The V-diagram [36] is a life-cycle model for software development that emphasises the relationship between the design, integration and testing processes. One such V-diagram is shown in Figure 2.5.

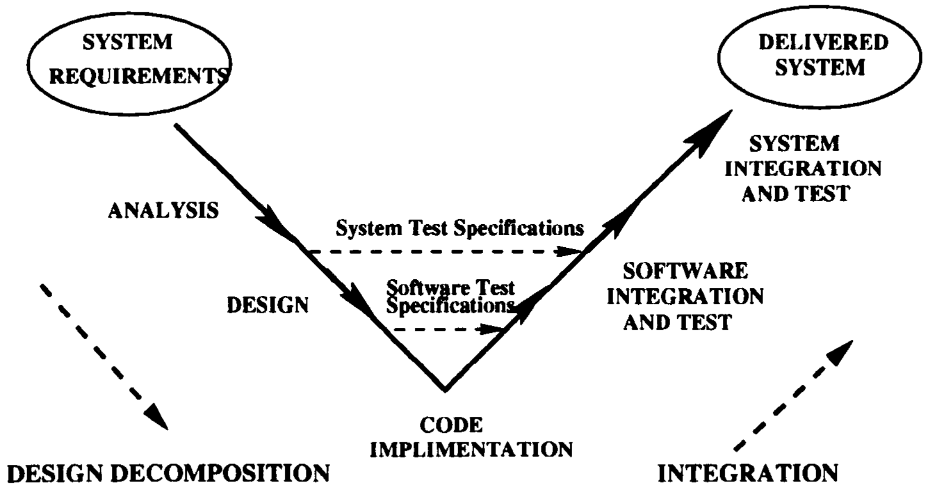

SYSTEM REQUIREMENTS
DELIVERED SYSTEM
SYSTEM INTEGRATION AND TEST
ANALYSIS
System Test Specifications
DESIGN
Software Test Specifications
SOFTWARE INTEGRATION AND TEST
CODE IMPLIMENTATION
DESIGN DECOMPOSITION
INTEGRATION

Figure 2.5: The V Diagram

In the V-diagram, the level of detail increases down the page. The left leg of the "V" shows the design and building of the system. The right leg shows the cross-checking and integration of the system. The diagram particularly emphasises the relationship between the phase of integration and the phase of design which provides the source of information for cross-checking. This is shown by the dotted lines.

A key issue for most systems, especially safety-critical systems, is the approach taken to demonstrate that the system will operate when it is delivered. The V-diagram approach divides the responsibilities for correctness into the two following forms :

- The progressive demonstration of consistency of the deliverables on the left leg of the "V", known as verification. Methods such as prototyping, proof of properties and reviewing are used here.
- Cross-checking during the integration phases of development, on the right-leg of the "V", establish that the delivered system meets the requirements

established at the corresponding phase on the left leg of the "V". Methods such as testing are used here.

### 2.1.2 Software Development Strategy

We will use the term "Software Development Strategy" to mean an elaborate and systematic plan of the activities involved in the software development process. This strategy may or may not involve formal methods.

We classify strategies for development as follows :

- **Structured method strategy**

  This is a strategy for development involving systematic methods for the analysis and design of complex systems. These methods can be contrasted with more ad hoc approaches, which are largely based on the designer's experience and intuition. There are many types of structured analysis. Most of these are semiformal, operational notations closely related to data flow diagrams (DFDs), a graphical notation used to describe the structure of an information system.

  The systematic software design methods make use of graphical representation forms, supported by varying degrees of structured text and free text. One problem with most of these notations is that they generally lack any rigorous syntactic and semantic foundations, and so it is difficult to reason about them in any 'formal manner'. Jackson's Structure Diagrams [36, 85], which are used to show both program and data structures, have a well-defined syntax but only some semantic content while Statecharts and Petrinets are more rigorous.

  The problem with this lack of firm-syntax and well-defined semantics for many of the diagrammatical notations used in systematic design practices is that no design verification is possible. So, it will become virtually impossible to make a comparison between the eventual design and the initial requirement. Hence a development strategy incorporating formal methods offers advantages.

- **Formal method strategy**

  This is a development strategy incorporating a set of techniques and tools based on mathematical models that are used to specify and verify requirements and designs for computer systems and software [120, 53, 95]. Such mathematical techniques and tools constitute **Formal Methods**. They employ formal specification languages based on mathematical structures. The use of such formal languages permits the application of mathematical techniques in reasoning about a design and its properties.

  Although a number of formal methods are well developed, their industrial use has been limited so far but is undoubtedly growing. Some of the reasons for the slow adoption of formal methods are the conservative approach of many project managers, the reluctance of customers to accept 'unfamiliar' techniques and notations, the need for familiarity with logic and discrete mathematics and the lack of adequate tool support.

- **Lean formal method strategy**

  This is a development strategy where formal methods are involved in such a way that it accommodates a wide spectrum of users. Hence, **Lean Formal Methods** are Formal Methods which are more accommodative to a wider spectrum of users. The strategy aims to make users conveniently use and/or understand formal methods as explained below. Central to this strategy is the provision of means to more comprehensible and user-friendly formal specifications and an associated development technique within a single framework encompassing refinement[1], abstraction[2] and animation of specifications[3]. It is important to note that the lean approach does not compromise on formality ; it maintains the preciseness involved in a formal approach but incorporates new techniques to bring

[1]see chapter 5
[2]see chapter 5
[3]see section 4.3

the formal approach closer to the user. Such lean techniques could involve automation to perform several analyses on specifications, like proving desired properties, the use of visual languages and so on, to enable better accessibility. This approach is in contrast to approaches where researchers try to give a formal semantics to a visual modelling framework like STATEMATE [69], Stateflow [11] or the Unified Process involving UML [33].

## 2.2 Specification

System specification is the most crucial activity in software projects. Quite often, massive overruns in budget and project duration are due to errors in describing the required system behaviour [82]. Understandably therefore, this activity, amongst all the activities in a software project, has been receiving the most attention over the last two decades.

Developing a complex software product requires the developer to carry out several software activities including system specification, system design and programming apart from testing and verification activities. System specification is the process of describing *what* a system is supposed to do. In order to develop the system specification, the developer is provided by the user with an informal statement of requirements.

System design is the process of using the software specification and developing an architecture expressed in terms of their modules and their interfaces. The objective in system design is to develop a description of *how* the system should operate rather than describe *what* is required. We can note here, again, that describing what behaviour is required to be exhibited by the system is the objective of the system specification. Programming is the process of translating a system design into program code.

A number of reasons make the task of specifying a system difficult [82]. Firstly, the informal statement of requirements given to the developer suffers from a number of deficiencies. In general, such statements are contradictory; incomplete; ambiguous; sometimes over-ambitious; and sometimes under-ambitious. They contain descriptions of functionality at different levels intermixed together. Although these deficiencies are normally associated with the user's statement of requirements, many of them crop up in system specifications. The second reason for the specification task being difficult is the nature of the notations used to express system functionality. English has been used widely as a language for system specification. However, English, or any other natural language, is an excellent medium for activities such as writing poems where ambiguity is regarded as the norm. This property of ambiguity leads to many problems when such natural languages are used for specifying systems. The seemingly innocuous sentences in such descriptions in natural language could be interpreted by the system designer in a manner completely different from that which the customer intended. Another problem with system specification is the size of the documentation generated. The specification of a simple stock control system for a retailer can occupy hundreds of pages of text, while the specification of a conventional engineered product, say a bridge, might occupy only two or three pages of text.

Until the late sixties, developers produced programs as though the process was more of an art than science and engineering. The early specifications were expressed in English and, although this was an improvement over not writing down system functionality at all, it lead to the problems described above. Graphical notations for system specification, after a slow start, have now picked up in software projects. Notations associated with structured development methods such as Yourdon Structured Design [152], SSADM [43, 5] etc. have gained widespread popularity for large-scale software development. At the most extreme end of the spectrum of specification notation is mathematics. It represents a radical change from current approaches. Mathematics has had a mixed impact on software projects, and has the longest history as a specification notation. In 1948, the English computer scientist Alan Turing used logic, the mathematics of true and false statements, to define the action of an assembler subroutine [82]. In the late 1960s, a large amount of research was carried out into program proving, instigated by the English computer scientist Tony Hoare [82, 80] and the Dutch researcher Edsger Dijkstra [82, 80]. The aim in program proving is to describe the function of a piece of program code using mathematics. The aim was to write the program and then prove - without testing - that the program meets its mathematical specification. In the 1960s, testing was a more difficult process than it is now[4] as there were virtually no debuggers, no dynamic analysis tools, and no facilities to store and rerun tests. Consequently, any alternative to testing - particularly one that required no testing effort at all - was treated very seriously. Unfortunately, a number of problems were discovered in program proving. The major one was that the process of proving that a program meets its specification was long and tedious and, more often than not, gave rise to proofs which were textually very much longer than the program itself. In order to overcome this problem, researchers attempted to develop automatic program provers : software tools which processed a mathematical specification and a program, and decided, with little human intervention, whether the program was correct. The search for an efficient program prover has been unsuccessful. Although the failure of automatic program proving lead to a diminution of research into formal methods of software development, the early eighties saw a dramatic resuscitation for two reasons. One reason was an increasing dissatisfaction with development performance on medium to large software projects. The other was a programme called the Alvey programme [82] whose software engineering component was, largely, concerned with formal methods of software development. What the Alvey programme indicated was a subtle shift in emphasis. Program proving was concerned with developing a specification and a program independently, and then showing that there was a match between them. The formal methods envisaged by the Alvey programme tended to concentrate on the specification first. Once a mathematical specification had been developed, the program would be constructed using the specification as a guide. The program development was carried out in small chunks, say a subroutine at a time, so that the complexity associated with program proving would be eliminated. This process, called *refinement*[5], will be illustrated in chapter 5.

[4]Today, testing is still difficult because effective test cases have to be designed, a lot of time has to be spent on testing besides managing tests more conveniently

[5]'Refinement' can be thought of as the process of elaborating on what is specified abstractly towards a specification that is executable.

Among the formal methods of software development currently being used on real projects are VDM (Vienna Development Method) [16] and Z [83]. The developer who uses VDM first identifies the objects and operations that occur in the application to be computerised. For example, in an air traffic control system typical objects would be planes and radars. Typical operations would be the creation of an aircraft when it came within radar range, the deletion of an aircraft when it landed and the reading of a radar signal from a particular radar installation. Once the objects and operations are identified, they are specified. Specification in VDM consists of defining the objects using a branch of mathematics called set theory which deals with collections of objects. Once the objects have been defined, the effect of operations is clarified. The effect is specified by means of a pre-condition and a post-condition. The former describes what must be true before an operation is executed ; the latter describes what must be true after an operation has been executed. Both these conditions are expressed using first order logic, to deal with the truth and falseness of statements.

Another popular formal method is Z. Currently Z is just a specification notation rather than a development method. It is semi-graphical in that it consists of mathematics enclosed in boxes known as schemas. Each box describes some stored data and the effect of operations on that stored data. Its advantage, compared with VDM, is that it provides a much more modular description of a system. This enables staff concerned with design, and with requirements analysis, to enable only those parts of the specification with which they were concerned. The disadvantage of Z, as compared to VDM, is that it lags behind in terms of facilities for transforming the notation into program code.

There are a number of reasons for the resistance encountered to the attempts of formal methods to gain a widespread usage. Despite their need, especially for safety-critical systems, there does not exist an integrated suite of software tools and techniques, formal or non-formal, that can be used during the various stages of system development. There is a growing need for systems that are provably correct, especially if the systems are real-time and safety-critical, and for such systems, the development of an integrated suite of tools and techniques together with an associated strategy is essential.

## 2.3 Real-Time Safety-Critical Systems (RT-SC) Systems

A **Real-Time System** is defined as a system which is capable of reacting to external events as they happen [101, 91]. Therefore, the computations of a real-time system not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. Therefore, if the timing constraints of the system are not met, system failure is said to have occurred. Common examples of real-time systems include flight control programs, air traffic control systems, control systems for power plants, patient monitoring systems, weapons systems, and military command and control systems. To be acceptable, such systems must not only be functionally correct but also be temporally correct. Before we develop any program/system, we would have to start with a description of its intended behaviour[6]. Such a description of the program/system is nothing but a 'specification' as described already and we noted that a natural language like English, though convenient, is very imprecise. Therefore, logic, which can lead to an unambiguous description of the requirements, becomes most suitable. In order to specify systems with timing constraints, a logic which has temporal constructs is the most suitable one. Such a logic is called 'temporal logic'. Among other methods to specify and reason about real-time systems are state machine models and models that extend process algebra. Most logics designed to reason about real-time systems are either first-order logics or temporal logics.

A system is safety-critical if the occurrence of unintended events could result in death, injury, illness, or damage to, or loss of, property. To avoid entering an unsafe state, a safety-critical system must often perform a given action by a specified deadline. Most (if not all) safety-critical systems are also real-time systems. For example, an air

[6]The 'behaviour' of a system is the development of states and state transitions generated by actions of the system during the time interval in which it is studied.

traffic control system is a real-time system with many stringent timing requirements and since the failure of such a system to act in time can have disastrous consequences, it is a safety-critical system also. Such systems are often complex with behaviours that depend on inter-relationships among the timing of the events of the system. Often, testing of such systems is inadequate as the sole means for ensuring their correctness.

## 2.4 Specification Methods for Real-Time Safety-Critical Systems

This section introduces specification methods for real-time, safety-critical systems.

### 2.4.1 A Note on Different Levels of Formality

Many methods have both formal and informal components. Therefore, it is quite common to talk about semiformal methods, that is, methods that are only partially defined in a precise, mathematical way. Some aspects of the method are left undefined ; understanding these aspects requires intuition and common sense. In many cases, semiformal methods define the syntax of a given notation rigorously but leave the notation's semantics undefined.

### 2.4.2 Models Based on Logic

**Some historical perspective on logics**

Early work on logic-based program verification includes work by R.W.Floyd [50] in 1967 and C.A.R.Hoare [77] in 1969. General surveys on the role of temporal logic in computer science include Pnueli [126], Goldblatt [60] and Emerson [44]. A temporal-like calculus for the specification and reasoning about concurrent programs was first proposed by Pnueli [127], and a temporal semantics for reactive programs was also proposed by Pnueli [128]. Some of the earlier applications of temporal logic for the specification and verification of concurrent programs are reported by Hailpern [65], Hailpern and Owicki [66], Owicki and Lamport [123], and Lamport [99]. More details and references can be found in [107].

Using logic, one can describe and reason about the behaviour of a system without building the system first. Such reasoning could be based on correctness proofs and/or aided by prototyping if the logic specification is executable.

The most widely known logic is first-order logic. For specifying a power production plant, in a first order theory, the plant and its properties can be described as follows. Variables such as *Temp* for temperature, *Pres* for pressure and *PowerRequested* for the amount of power requested are used to express all relevant system quantities. The variable $t$ is used to represent the critical system component of time. Predicates are used to specify system properties and constraints. For example, the following predicate $PowerProduced \leq PowerRequested$ states that the power produced cannot exceed the requested power. The predicate can be rewritten to show the time dependence of the variables as :

$\forall(t) PowerProduced(t) \leq PowerRequested(t)$

In the power production plant example mentioned above, if we require a signal *ModerateDanger* to be raised when the temperature and pressure have been continuously above the limits $T_{max}$ and $P_{max}$ respectively for D time units, then we can use the following :

$\forall t[\forall t', t-D \leq t' < t : Pres(t') > P_{max} \wedge Temp(t') > T_{max} \rightarrow ModerateDanger(t)]$.

Using such formulas, we can describe the required behaviour of the power plant completely. For example, requirements such as the following can be specified :

- The system must be shut down within h time units after the *HighDanger* alarm sounds.
- The amount of coolant injected into the plant is proportional to the product of the difference between the actual temperature and the ideal temperature times the difference between the actual pressure and ideal pressure. However, if there is no more coolant in the tank, the *HighDanger* alarm sounds immediately.

The logic notation, unlike the SA/RT notation, can describe the above requirements and similar functional, control, or timing requirements completely and precisely.

The logical approach can be used to formalise basic system properties as axioms, i.e., fundamental facts that by assumption are guaranteed, and to derive additional properties as theorems, consequences that follow from the basic assumptions. As an example, suppose the following two axioms are assumed :

1. Once the plant is turned off, temperature and pressure decrease with time according to some given mathematical function.
2. As soon as temperature and pressure are again within safety limits, the plant is immediately restarted.

Using these axioms, we can prove as a theorem that the plant will never be off for more than some maximum time.

Since first-order logic is a basic mathematical formalism, it is not well suited for describing real-world complexities. Thus, in practical systems, logic is most useful for proving critical system properties rather than proving properties about the whole system. Another complication in the practical application of first-order logic is that sufficiently general theories are undecidable; that is, no algorithm exists that can determine whether a given property can be proven as a theorem.

Although time, in principle, can be incorporated into a logic in the same way as other system variables, many researchers advocate a special role for time and have introduced new logics for reasoning about time. Some are first-order logics whereas others are temporal logics. A detailed discussion of one such temporal logic called ITL will follow in chapter 4.

Some examples of first-order logics designed for reasoning about real-time are Real-Time Logic or RTL [88], TRIO [57] and ASTRAL [35]. A special feature of TRIO is that a special interpreter makes specifications in TRIO executable. ASTRAL, another logic-based language, combines the timing features of TRIO with the structuring mechanisms of ASLAN [6], an earlier language for describing non-real-time systems. ASTRAL specifications describe a system in a fairly operational style by defining its states and state transitions. Proof obligations are built to drive the formal analysis of the specification in a deductive style. The system specifications are treated as a set of axioms, from which the system properties are derived as theorems. Though presently no tools are available to support ASTRAL proofs, in principle, semiautomatic tools that generate proof obligations from a given specification could support formal analysis in a manner similar to that previously demonstrated for ASLAN.

Most real-time logics describe systems in terms of events, points in time when something significant happens. In contrast, a few others called interval logics, focus on conditions (e.g., states) that hold for some nonzero time interval. Allen describes a classical example of an interval logic. The Duration Calculus [153, 154], a real-time formalism, is a special case of interval logic. Section 2.6 explains the rationale behind the choice of Interval Temporal Logic for this work.

### 2.4.3 State Machine Models

A state machine is a mathematical model of a system with input and output. A finite state machine (FSM) is a state machine with a finite number of states only. Formally, a finite automaton can be defined as a five tuple $(Q, \Sigma, \delta, \lambda, q_0, F)$ where

Q is a finite set of states,

$\Sigma$ is the input alphabet,

$q_0 \in Q$ is the initial state,

$F \subseteq Q$ is the set of final states

$\delta : Q \times \Sigma \rightarrow Q$ is the transition function.

Section 3.1.6 gives additional information on the state machine.

**Timed Automaton Model**

This model [106, 110] is based on dense time. It allows infinitely many states unlike the classical state machine model which has a finite number of states. The model describes a system as a collection of automata (i.e., state machines) interacting by means of common actions. Important actions in the model are input and output actions, both of which are visible outside the system. Time is added to the model through a special time passage action.

The state transitions are described by specifying the "preconditions" under which each action can occur and the "effect" of each action.

To add time to the model, time bounds are associated with each action. With this technique, variables as well as constraints may be used to represent timing information. Representing time with variables allows constraints on variables to be derived later on from other information in the specifications.

The timed automaton model overcomes the limitations of the classical state machine as time is built into the state. Also, general logic formulas may be used to specify transitions.

To solve a problem using the timed automaton model, two system descriptions are developed, one specifying the problem, the other the solution. Proofs are constructed to show a "simulation mapping" between the two descriptions. If every behaviour of one description is a behaviour of the other, then the simulation mapping between the two descriptions holds. The two specifications are equivalent if the two sets of behaviours are equal.

**Esterel family**

Esterel is the most widely known member of a family of languages that uses the state machine model to describe real-time systems. The Esterel family is based on the *synchrony hypothesis*, which states that each system's response to a set of inputs is instantaneous. At the practical level, this means that the system must complete all computations before the next input from the environment arrives. The website [46] is a good reference for obtaining the compiler, related tools and additional information.

The Esterel family places emphasis on the later phases of the software life cycle unlike many other real-time formalisms. Compilers are available which automatically produce running code from specifications written in the language of some family member.

Esterel and other members of the family have been applied to many industrial projects, mainly in the fields of nuclear safety and avionics.

A comprehensive description of the synchronous approach and the Esterel family of languages is given in references [14, 67].

### 2.4.4 Process Algebras

CCS [78], CSP [111], and ACP [13] are process algebras originally developed to specify and analyse concurrent systems without the notion of time. A process algebra has a concise language, a precisely defined semantics, a notion of equivalence or preorder, and a set of algebraic laws allowing syntactic manipulation. The language is based on a small set of operators and a few syntactic rules for constructing a complex process from simpler processes. The semantics describe the possible execution steps a process can take. Two processes are equivalent when they have the same behaviour, that is, when every execution step of one process is also an execution step of the other process and vice versa. A preorder between two processes exists when the behaviour of one process is a subset of the behaviour of another. To verify a system using a process algebra, one writes a specification as an abstract process and an implementation as a detailed process. To prove correctness, the two processes are shown to be equivalent or a preorder between the processes is shown. The proof of correctness is accomplished by syntactically manipulating the algebraic laws.

A number of timed algebras have been proposed recently. These include ACSR [22, 100], which adds time to CCS ; Timed CSP [132], a timed version of CSP ; Timed LOTOS [18], a timed version of the ISO standard LOTOS, which is also based on CSP and has already been applied to several industrial projects ; and a timed version of ACP.

## 2.5 Analysis Techniques for Real-Time Safety-Critical Systems

Simulation, model-theoretic and proof-theoretic reasoning are major classes of techniques developed for formally reasoning about real-time systems.

### 2.5.1 Simulating/Executing Specifications

Many formal methods researchers underestimate the value of simulation in exposing defects in specifications. Using simulation, the user can ensure that the behaviour specified satisfies his/her intent. Unlike consistency checking, model checking, and mechanical theorem proving, which formally check the specification for properties of interest, simulation provides a means of validating a specification. In [72], Heitmeyer, C., mentions that a powerful, customisable simulation capability is very much necessary in the formal development process. In running scenarios through the simulator, the user can also use the simulator to check properties of interest. Another use of simulation is in conjunction with model checking; counter-examples obtained from a model checker can be fed to a simulator to demonstrate and validate them.

### 2.5.2 Model-theoretic Reasoning

A number of algorithms have been developed in recent years to verify properties of systems modelled as state machines. One class of algorithms, called model checkers, were invented by Clarke and Emerson in 1981 [34, 80] to verify properties of untimed specifications. These algorithms take a finite state machine model of a system and a temporal logic formula and determine if the formula is true of the model. While design errors in practical systems have been detected using model-theoretic reasoning, the errors were in untimed specifications. The application of these techniques to timed specifications is a significant area of current research. Scale is a problem in the verification of specifications of real-time systems as the addition of time to system specification produces a model which is usually too large to analyse.

For example, the Modechart verifier [89] is designed to analyse real-time specifications using model-based techniques. This tool builds a *computation graph* to represent all possible states that a system can enter. Various approaches are used to prune unreachable nodes in the graph and to combine duplicate nodes. If the number of states and the timing constants in the model become very large, the computation graph becomes too large to build and analyse. Therefore, the Modechart verifier and similar tools can only verify small real-time models.

Techniques to handle dense time also exist. One promising approach uses approximations to avoid dealing with timing information in a specification until it is needed.

This method is more practical compared to the proof-theoretic reasoning because of its "push-button" nature. However, the proof-based methods can be made more practical by automating the process as much as possible and by providing built-in proof strategies.

### 2.5.3 Proof-theoretic Reasoning

In this technique, a theory about the system of interest has to be developed based on some logic. System properties are represented by formulas in the logic. Logical deductive techniques are applied to construct required proofs about the system. This requires mathematical maturity and theorem proving skills.

Developing proofs will require a lot of effort and time. However, the deductive approach has some advantages over model-theoretic reasoning. First, the user will gain a deep understanding of the system specification and its properties while developing the proofs. Second, more abstract models can be handled in proof-theoretic techniques and hence, more general results produced. Further, deductive reasoning can be applied to any mathematical model unlike model checking and other similar techniques.

A number of mechanical proof systems have been developed in recent years to support deductive reasoning. These include the Boyer-Moore theorem prover [21], -EVES [98], and the Larch Prover [64], all based on first order logic, as well as HOL [63] and PVS [124], which are based on Higher Order Logic. Such systems can do some proofs automatically. Most nontrivial proofs, however, require user guidance ; the user must develop the overall proof strategy. Mechanical proof systems can be very useful in checking such hand proofs. For safety-critical systems, a proof system that validates each step in the formal reasoning is very important as it would lead to an increase in the proof's validity as compared to a more informal proof which was not checked mechanically.

## 2.6 On the Choice of ITL for this Work

In the previous sections, we have explored various kinds of formalisms. This work is based on a logic-based formalism called Interval Temporal Logic (ITL) [84, 116].

ITL is a flexible notation for both propositional and first-order reasoning about periods of time found in descriptions of hardware and software systems. It can handle both sequential and parallel composition unlike most temporal logics. It offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and projected time. Tempura provides an executable framework for developing and experimenting with suitable ITL specifications. Hence, this work uses ITL as its formalism and aims to explore approaches which may enhance its uptake by a wider spectrum of users.

## 2.7 Drawbacks of Formal Methods

There are many formal methods, each having their own notation and varied degree of learning curves. This fragmentation and lack of standards further hinders their uptake. This is particularly so when the current software development practices do not involve formal notations. Real world projects tend to be large in scale, which raises the issues of scalability and communication of the chosen development technique. Formal methods neither support scalability nor ease of communication amongst members of a large team. Current work on compositional issues, which help in scalability, are still in their infancy. In addition, there is often a need for integrating many formalisms. For example, process algebraic styles of notation may be suitable for describing system interaction but they fail to describe the internal behaviour of a system. When many formalisms are involved in a development process, the user gets confronted with many learning curves.

The popularity of structured methodologies is often attributed to their associated graphical notation. What is required is a *compositional* formal method which enjoys both the preciseness of mathematical arguments and ease of communication associated with structured methodologies.

Towards that goal, various strands of development have taken place. Integrating formal methods such as Z [86] with structured techniques such as SSADM [5, 23, 43] have been attempted but with little success. This is mainly due to lack of scalability and the continuing presence of formal notation. In addition, various formal notations have been integrated, for example CSP and Temporal Logic. This type of integration did not alleviate the limitation of formal methods. Instead, the developers were faced with learning various formal notations each with their own semantics domain and verification style.

Another strand of development was directed towards executable/animatable specifications. This has met with some success as now formal notations look like a programming language with which developers are more familiar. However, issues such as scalability, learnability and being industry-strength remain unsolved.

It is therefore believed that a *lean formal* approach which combines a compositional graphical notation, animation and a development process within a single formal framework will overcome some of the fundamental problems associated with the use of current formal methods. Such an approach, by integrating various activities that constitute a development process in one single framework, would eliminate the burden of the user in terms of the "learning curve for formal methods" that he/she has to go through. In other words, this approach brings the formalism closer to the user and thereby encourages the user to adopt a formal development strategy in systems development.

## 2.8 Summary

In this chapter, we first explored software development process models i.e., Waterfall model, Prototyping based model, Iterative enhancement model, Spiral model, V-diagram model and the Transform model. We then used the term "Software Development Strategy" to mean an elaborate and systematic plan of the activities involved in the software development process and distinguished between three possibilities i.e., Structured method strategy, Formal method strategy and Lean formal method strategy. It was noted that structured methods had the advantages of being graphical in nature but lacked the advantages of being a formal notation. Formal methods, on the other hand, were very precise but had accessibility problems. A lean formal method strategy was defined as a strategy where the Formal Methods involved are more accommodative to a wider spectrum of users. Central to this strategy is the provision of means to more comprehensible and user-friendly formal specifications and an associated development technique within a single framework encompassing refinement[7], abstraction[8] and animation of specifications[9]. It is important to note that the lean approach does not compromise on formality; it maintains the preciseness involved in a formal approach but incorporates new techniques to bring the formal approach closer to the user. Such lean techniques could involve automation to perform several analyses on specifications like proving desired properties, the use of visual languages and so on to enable better accessibility. We then looked at what a specification is and why it is a crucial step in the software development process. We then examined specification methods for real-time safety-critical systems i.e., models based on logic, state machine models and process algebras. Analysis techniques like executing specifications, model-theoretic and proof-theoretic reasoning were explained. We stated the rationale for the choice of *Interval Temporal Logic (ITL)* as the formalism for this work. We also stated the drawbacks of formal methods, in general, and how a lean approach attempts to overcome these

[7]see chapter 5
[8]see chapter 5
[9]see section 4.3

hurdles to wider accessibility.

In the following chapter, we shall explore the nature of visualisation and its role in this work in the context of increasing the uptake of ITL as a formal development technique. In this context, we shall also examine existing graphical notations in specifications.

# Chapter 3

# Visualisation, Visual Languages and their Design Rationale

The main objective of this chapter is to see how visualisation helps in various domains in making information, languages and technology accessible.

## 3.1 The "What and Whys" of Visualisation

Visualisation provides a tool for achieving a clear mental image of the object under study. Whether it is an abstract mathematical transformation, census data or a nicely rendered Computer Aided Design part, graphics allows us to see the object of our interest more clearly than we could by other techniques. Graphics provides tools for visualisation and hence understanding of objects of interest.

Visualisation provides enhanced communication at the human/machine interface. The following examples illustrate the power of communication through visual images.

### 3.1.1 Public Signs

Figure 3.1 shows several iconic signs that are becoming fairly standardised in international usage, along with their meaning in English. The signs convey information in a language-free mode. The eyes are instinctively drawn to the iconic symbols first and only later begin to browse the natural language interpretation. The iconic symbols require low-level cognitive processing. Alphanumeric symbols, however, are compound symbols, made up of sentences and words comprising of strings of more elementary symbols (letters) that require a higher level of processing to extract their meaning.

In summary, it can be noted that intuitive iconic signs convey information to people across barriers such as language, culture and levels of literacy.

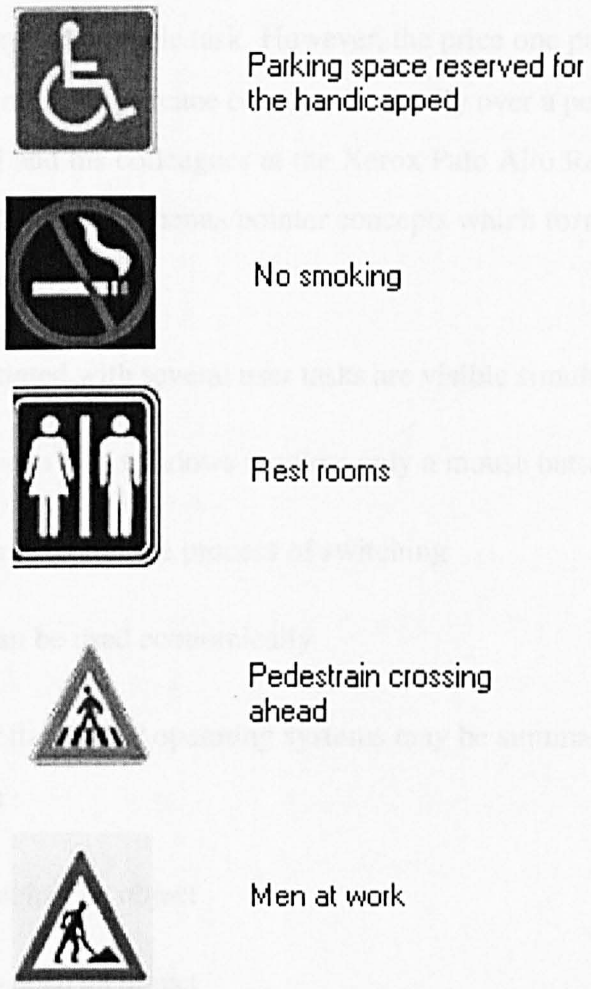


Figure 3.1: Iconic Public Signs

### 3.1.2 Operating Systems and Programming Environments

This provides an example of the extremes possible on the textual/visual spectrum. Traditional command-line languages such as MS-DOS and Unix fall at the textual end whereas Windows/Icons/Menus/Pointer (WIMP) interfaces, exemplified by the Macintosh operating system and Microsoft Windows on MS-DOS systems, fall at the visual end.

A main advantage of textual operating systems is the power and flexibility they provide in performing any conceivable task. However, the price one pays for this power is the effort required in mastering arcane commands usually over a period of one or more years. Alan Kay [93] and his colleagues at the Xerox Palo Alto Research Centre conceived the basic windows/icons/menus/pointer concepts which form part of the WIMP paradigm. Its basic principles are :

- Windows associated with several user tasks are visible simultaneously
- Switching between task windows requires only a mouse button push
- Information is not lost in the process of switching
- Screen space can be used economically

The operations of the WIMP operating systems may be summarised by the following three simple rules :

1. Single click to select an object
2. Double click to open an object
3. When in doubt, scan the menu for the appropriate function or help.

The knowledge of how to interact with and operate such systems is an intrinsic part of the system itself. In conventional command-line operating systems, however, this knowledge is archived in huge reference manuals. A WIMP system user does not have to painfully memorise or reference any manual, the system can provide such help on-line. People unfamiliar with computers can learn to operate and become productive with WIMP systems in a matter of hours or days compared to weeks or months of training required for the more conventional command-line systems. Experienced WIMP users often take great pride in exploiting the full capabilities of a totally unfamiliar software system without either memorising arcane mnemonics like "copy a:*.* c:" or opening a user manual. The combination of an intuitive, easy-to-visualise desktop environment has resulted in a dramatic decrease in training times and costs with a corresponding increase in productivity. These practical, "bottom-line" results have led to a rout of command-line operating systems as evident in the rush towards GUI environments like Microsoft Windows, X Windows and so on.

One example is the GNOME which is an acronym for the GNU Network Object Model Environment. It is a user-friendly desktop environment that enables users to easily use and configure their computers. GNOME includes a panel (for starting applications and displaying status), a desktop (where data and applications can be placed), a set of standard desktop tools and applications, and a set of conventions that make it easy for applications to cooperate and be consistent with each other. Users of other operating systems or environments should feel right at home using the powerful graphics-driven environment GNOME provides. GNOME has a number of advantages for users. GNOME makes it easy to use and configure applications using a simple yet powerful graphical interface. GNOME is highly configurable, enabling the user to set the desktop the way he or she wants it to look and feel. More details on GNOME can be found in [59].

KDE is a powerful graphical desktop environment for Unix workstations. It combines ease of use, contemporary functionality and outstanding graphical design with the technological superiority of the Unix operating system. More details are available at [94].

The Common Desktop Environment (CDE) is a commercial graphical user interface for Unix in its various flavours (AIX, Digital Unix, HP/UX, Solaris etc.). It is built on existing technologies from several Unix vendors (HP, Sun, IMB, Novell) and the Open Software Foundation Inc. (OSF). Based on Motif and X11, it is visually appealing and offers many productivity features. The CDE was created by this group of Unix vendors to consolidate all the Unix desktop interfaces and define a consistent user and development environment. Standardising the user interface enables general users and system managers to work more effectively on systems from all vendors that provide CDE.

To summarise, it can be stated that the use of user-friendly graphical environments have facilitated the interaction with and use of computer systems. A wide spectrum of users can be allowed accessibility with the use of such powerful visual environments. Such convenience to the user, particularly in terms of hiding the user from knowing intricate details of textual commands, helps in wider acceptance of such systems.

### 3.1.3 Visual Programming and Program Visualisation

The success of the visual paradigm for operating systems has also encouraged its application to programming languages. A special issue of IEEE Computer on "Visualisation in Computing" [147] details progress in this area. Visualisation with respect to programming languages and environments can be subdivided into two areas which are, Visual Programming and Program Visualisation. Myers [119] states that program visualisation is where a "program is specified in a conventional textual manner, and graphics are used to illustrate some aspect of the program or its runtime execution". He describes visual programming as the ability "to specify a program in a two or more dimensional fashion". Visual Basic is one example of a graphical programming language. Pict/D [58] is an example of a graphical programming environment where icons are used for visual programming. The user can compose simple programs for numerical computations using the subsystems represented such as programming (a flowchart metaphor), erase (a hand holding an eraser), icon editor (a hand holding a pen) and a user library (a shelf of books). The user can program an icon, edit it, or run its associated program. The resultant program can be denoted by a new icon, created by the user with the icon editor, and stored in the library for future use. Further references to additional information on visual languages can be found in [31] and [139].

### 3.1.4 Computer Simulations

Computer simulations are valuable as research tools for modelling physical system properties that are too difficult, dangerous or expensive to measure. They are also of tremendous value to engineers in modelling mechanical, thermal, and electromagnetic systems by the techniques of Finite Element Analysis (FEA).

Once simulations are performed, good computer graphics and high speed 3-D processing are a necessity for post-processing. The ability to read selected results from analysis output database files is essential.

Another interesting area where visualisation plays a central role in engineering design is Computer Aided Design (CAD). CAD has been a major influential force in the development of sophisticated computer graphics techniques. CAD systems are now essential tools for the design of integrated circuits (ICs), manufactured parts, complex mechanical systems, and architectural plans. Advanced CAD systems permit the direct connection of the CAD design phase to the Computer Aided Manufacturing (CAM) phase. Design parameters from the CAD program are directly used for the numeric control of machine tools. This eliminates the stage of hardcopy blueprints and thereby contributes to an increase in productivity.

**Specification Diagrams**

Structured Analysis/Real Time (SA/RT), State Transition Diagrams, Statecharts, Statechart-like notations such as Modechart and Timed-Transition Models, Petri Nets, Graphical Interval Logic (GIL), Ladder Logic Diagrams and UML Diagrams are graphical notations for the specification of systems.

### 3.1.5 Structured Analysis

Before considering structured analysis, it is worth noting that "flowcharts" were one of the earliest methods of showing programs in picture form. Figure 3.2 shows some of the main symbols used in a flowchart and a simple example of a flowchart. A major criticism of the flowchart is that it mainly describes a solution in terms of the operations of the underlying machine, rather than in terms of the problem and its structures. It closely resembles the final program code in structure and hence is not much more abstract than the implementation. It also does not support structured design methods and hence was gradually replaced by design methods involving structured methods. Structured Analysis is a general term used to describe systematic methods for the analysis and design of complex software systems. These methods can be contrasted with more ad hoc approaches, which are largely based on the designer's experience and intuition. There are many types of structured analysis. Most of these are semiformal, operational notations closely related to data flow diagrams (DFDs), a graphical notation used to describe the structure of an information system. SA/RT (Structured Analysis/Real Time) is an enhancement of structured analysis designed to model real-time systems. Major constructs of SA/RT are Transformation Schemata (TS), an extension of traditional DFDs.

SA/RT provides refinement mechanisms useful in building large, well-structured specifications. Teamwork and Software Through Pictures are among commercial tools compatible with SA/RT. Such tools however are mainly documentation tools and cannot support semantic analysis. However, they prove useful for many industrial projects.

Recently, object-oriented versions of structured analysis methods have been proposed. Some of these are Shlaer and Mellor [138], HOORA [45] and ROOM [137] and are meant for developing real-time systems.

### 3.1.6 State Transition Diagrams

State transition diagrams originate from the theory of automata. A finite automaton is a mathematical model of a system with discrete inputs and outputs. The system is in

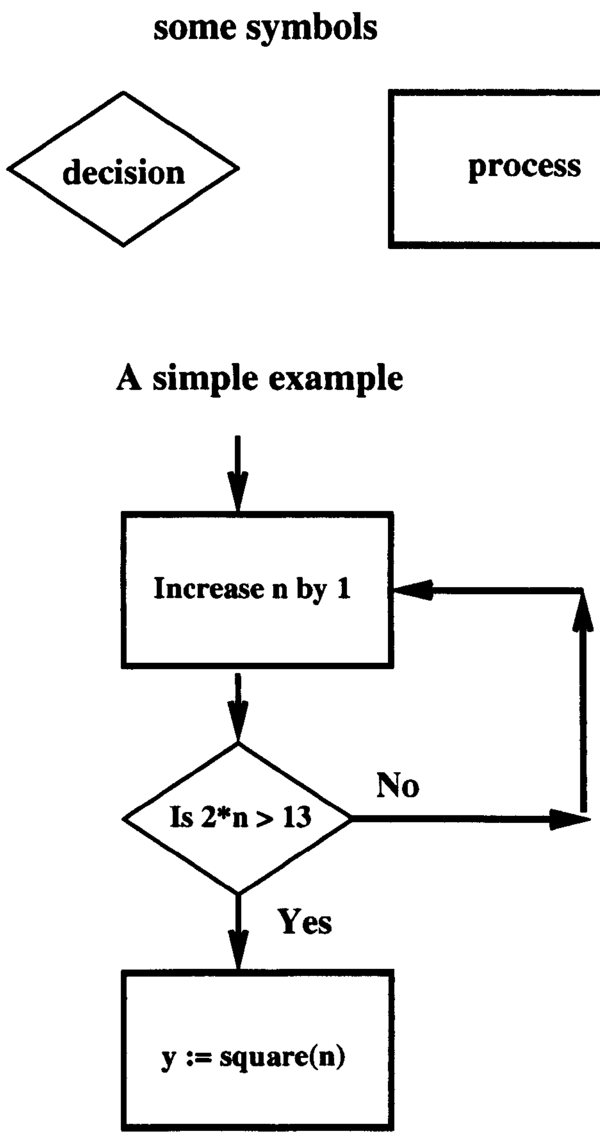


Figure 3.2: Flowchart

only a finite number of states. Such a state contains enough information over previous input to determine the behaviour of the system on the next input. The control system for an elevator can be represented as a state transition diagram, for example, in which not all requests are remembered but only the current floor, the direction of movement i.e, up or down and the not yet handled requests. Human brains can also be similarly described. However, these systems have such a huge number of states that treating them as finite state systems will serve no purpose. An example state transition diagram for an automaton is given in Figure 3.3. The automaton shown is in state $q_0$ to start with and then transitions to other states $q_1$, $q_2$ and $q_3$ based on the input it receives as shown. The state with the double circle i.e., state $q_0$ is also the final state for this automaton.

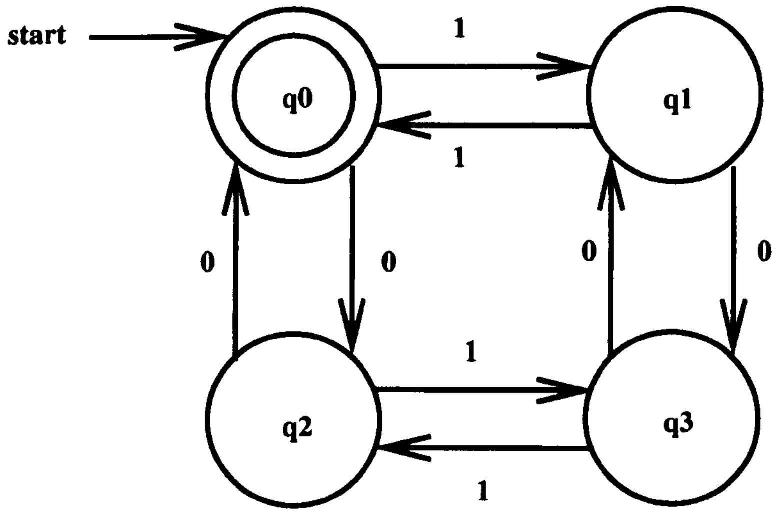


Figure 3.3: An Example State Transition Diagram for an Automaton

### 3.1.7 Statecharts

Statecharts [68], a graphical language for specifying real-time systems, exploits the naturalness and simplicity of the classical finite state machine. This notation also overcomes, to a large extent, the state machine's major shortcomings ; the most important is the combinatorial explosion in the number of states. A system which combines two machines, one with h states and another with k states, has a state space of h x k states, that is, the Cartesian product of the original ones. AND and OR composition are two constructs used in statecharts to overcome this problem. The AND composition is described below.

The AND of two component machines formalises the composition of two concurrent subsystems into a single aggregate machine. Let us consider a railroad crossing example to illustrate the AND composition. Let *Out*, *P* and *I* be the states representing the location of the train with respect to the crossing i.e., the location of the gate, where the state *Out* is to mean that the train is far away from the crossing, state *P* is to mean that the train is near the crossing and state *I* is to mean that the train is in the crossing.

Let $Up$ and $Down$ represent the states for the gate at the crossing.

The set of states in the composition of the two machines contains all possible combinations of the individual states, i.e., $< Out, Down >, < Out, Up >, < P, Down >, < P, Up >, < I, Down >$ *and* $< I, Up >$. The transitions between states are combined similarly : the set of possible transitions is the union of the original sets. In statecharts, the concurrent composition of the component machines is left implicit but formally defined through the AND construct, which composes the two components into a single, larger machine as shown below. Some state transitions may not be possible in the larger state

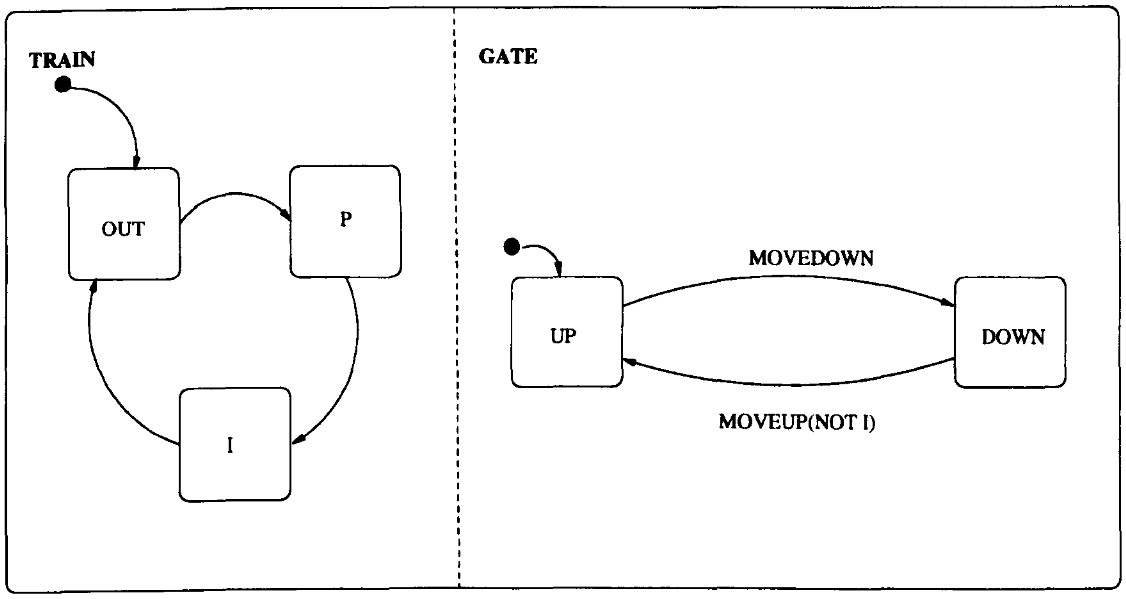


Figure 3.4: Statecharts Specification showing AND-composition of Train and Gate

machine because when the system is in a state in one of the smaller machines, it cannot enter a specified state in the second machine. One way to model the coordination of the two smaller machines is to label a transition with a formula. The transition can only be taken if the formula is true. For example, in Figure 3.4, attaching the formula "not in I" to the transition *MoveUp* specifies that the gate cannot be raised if a train remains in I.

Figure 3.4 above illustrates some important limitations of the classical state machine model for describing real-time systems :

- Time-dependent behaviour, e.g., "The gate must be down within 10 seconds after a train enters P", cannot be expressed.

- The figure captures only a small fraction of the required systems behaviour. For example, labelling *MoveUp* as described above prevents this transition when a train is in I but does not guarantee that the gate will be down whenever a train is in I because the gate could move up and remain there. More information is needed to describe the system in detail, such as when the gate must start moving down once a train has entered R.

- If, in a generalisation of the system, several trains can be in R simultaneously, then, in the classical model, each train must be described separately. As the number of trains grows, the model becomes unwieldy. Also, the model cannot handle an unbounded number of trains.

Statecharts have addressed these problems, at least in part. First, two simple classes of time-dependent behaviour can be expressed in Statecharts: a *timeout event*, an event scheduled to occur a fixed number of time units after another event, and a *scheduled action*, an operation (e.g., fire a missile) scheduled to occur a fixed number of time units after the current time. In addition, Statecharts allows parameterised states (e.g., "the ith train enters R") and the attachment of generalised formulas to transitions.

The commercial version of Statecharts, called STATEMATE [81, 68, 69], offers these features as well as others. STATEMATE has had considerable success in industry because it has a user-friendly interface that complements the intuitive appeal of the classical state machine formalism. Moreover, STATEMATE offers two forms of analysis:

- The user can run STATEMATE's simulator to analyse the behaviour of a "system model" in scenarios of interest.

- STATEMATE's Dynamic Tests tool can do reachability analysis. From the Statecharts specification, the tool builds a reachability graph containing possible states the system can be in. Using this graph, the tool can check for deadlock, nondeterminism and race conditions. It can also search for a reachable state in which a condition is true.

Like semiformal approaches, Statecharts provides documentation support but, unlike them, Statecharts also supports some semantic analysis. However, compared to some of the newer formal methods (e.g., model checking techniques), Statecharts' analysis capability, which is confined to using reachability analysis to check a small set of properties, is quite limited. Moreover, the ability to specify and reason about a system's timing behaviour using Statecharts is also quite limited. Quite recently, a model-checker for Statecharts has been developed [39].

We note again that Statecharts overcomes the limitations of state transition diagrams in an intuitive way with depth represented by the use of substates i.e., having a nesting of states, orthogonality represented by the partitioning feature using a dotted line and the use of broadcast communication.

### 3.1.8 Statechart-like Notations

Modechart [89] and Timed Transition Model [121, 122] borrow heavily from Statecharts but are more expressive than Statecharts for describing and reasoning about timing properties.

- **Modechart**

  Important constructs in Modechart include *modes*, which correspond to states in Statecharts, and *actions*, which assign values to data variables. A third Modechart construct is the *event*. Different types of events are external events, which represent changes in the system environment (e.g., the operator pushing the START button); mode entry and mode exit events, which mark entry into or exit from a mode; and start and stop events, which mark the beginning and end of an action. Deadlines and delays, upper and lower bounds on the time interval from mode entry to mode exit, allow the specification of the time that a system can remain in a mode. Modechart uses a discrete time model ; so, its delays and deadlines are represented as non-negative integers. In Modechart, events are instantaneous, whereas actions require at least one time unit to complete.

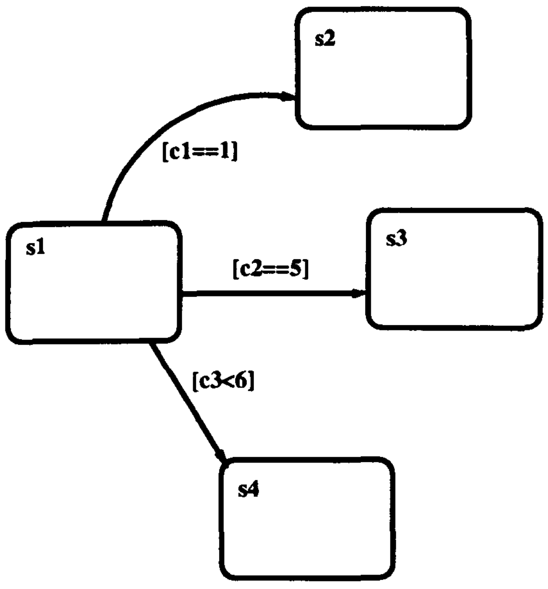
In Modecharts, modes may be serial or parallel. Modechart's notion of serial and parallel modes corresponds to OR and AND composition in Statecharts. If M is a serial mode with child modes $M_1$ and $M_2$, then at a given time the system is in exactly one of $M_1$ and $M_2$. If M is a parallel mode with child modes $M_1$ and $M_2$, then when the system is in M, it is simultaneously in both modes $M_1$ and $M_2$. Modechart's semantics are defined in terms of the real-time logic RTL [89].

- **Timed Transition Models**

  The activity variables (or activities) in a Timed Transition Model (TTM) specification correspond to states in Statecharts. A group of subactivities may be composed into an activity using the Statecharts notions of AND and OR composition. Other major components of a TTM specification are events and integer variables. TTM supports both the usual relational operators and integer arithmetic in contrast to Modechart which does not support arithmetic. As in Modechart, time in the TTM framework is discrete, and timing constraints are represented as lower and upper bounds on transitions between activities.

### 3.1.9 Stateflow Diagrams

Stateflow diagrams, like statecharts, also use a variant of the finite state machine [68]. A stateflow diagram is a graphical representation of a finite state machine where *states* and *transitions* form the basic building blocks of the system. Apart from these, a stateflow diagram also enables the representation of hierarchy, parallelism and history. One important feature of a stateflow diagram is that its execution is dependent on the geometry of the diagram. For example, if there are conflicts to resolve regarding transitions, they are resolved based on the geometry of the outgoing transitions as depicted in an example in Figure 3.5. In other words, non-determinism is resolved by resorting to geometrical considerations. This resolution based on geometry applies to transitions



**The three transitions are evaluated in a clockwise progression starting at the upper, left corner of state s1.**

**If the condition "c1==1" holds, then the transition to s2 is taken. Only if that condition does not hold is the condition on the the transition to s3 checked and so on.**

Figure 3.5: An Example of Resolving Non-determinism in Stateflow

which are equivalent with respect to their labels i.e., what the transitions are annotated with among the four following possibilities given in the order of priority.

1. Events and Conditions

2. Events

3. Conditions

4. No label

In Figure 3.5, the transitions were all equivalent as they all had only conditions. The non-determinism among those equivalent transitions was resolved by geometry. In stateflow, parallel states are depicted using rounded boxes unlike STATEMATE. For additional details regarding stateflow, the reader is referred to [143].

## 3.1.10 Petri Nets

A Petri Net [125] consists of a set of places, usually denoted as circles, and a set of transitions, usually denoted as bars. Arrows connect places to transitions and transitions to places. A place is called an input of a transition (a transition is called an input of a place) if there is an arrow going from the place to the transition (from the transition to the place). A similar definition holds for output places and transitions. The Figure 3.6 illustrates a simple Petri net. Petri nets are an operational formalism ; they support the

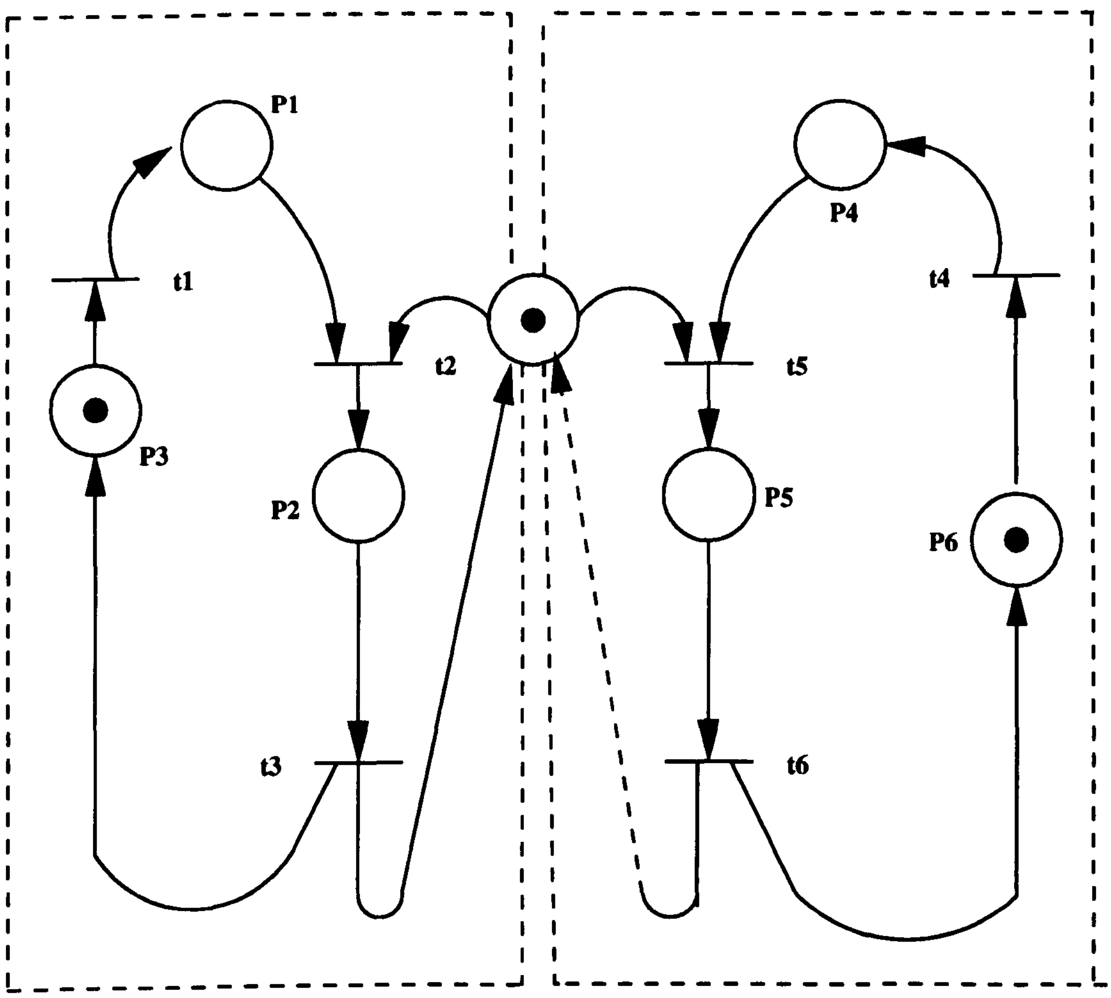


Figure 3.6: Example Petri Net

notion of a state and its evolution. The state of a Petri net is represented graphically as a marking of its places with an assignment of a nonnegative number of tokens to each place. The evolution of a Petri net occurs according to the following rules :

- A transition is enabled in a given marking if all of its input places are marked, i.e., each has at least one token.
- An enabled transition may fire. This means that one token is removed from each input place of the transition and one token is put into each output place thereof. Thus the firing of a transition produces a new marking.

Some of the problems that need to be addressed before Petri nets can be used effectively in designing real-time systems are given below followed by a summary of approaches that have been proposed over the years to solve these problems.

- Because basic Petri nets are designed to describe concurrency rather than the passage of time, they cannot express timeouts and durations.
- Petri nets lack abstraction mechanisms and thus become large and unmanageable when one moves from small examples to practical applications.
- Petri nets only model a systems's control features, not its data dependencies. Because tokens are "anonymous", dependencies between control and data cannot be modelled. For example, a rule, such as "uncorrupted message must be forwarded through channel 1, whereas damaged ones must be sent back through channel 2", cannot be described formally.

A number of approaches, some with tool support, have been proposed to solve these problems. Some of them are :

- Time has been added to Petri nets in many different ways. One of the most general extensions associates a minimum and a maximum firing time with each transition [109]. Once enabled, a transition cannot fire before the minimum time and must fire by the maximum time, unless previously disabled by the firing of a conflicting transition.

- Several abstraction and modularisation mechanisms, essential in modelling real-world systems, have been proposed to support the construction of hierarchical, well-structured Petri nets. Recently, such mechanisms have exploited the object-oriented paradigm.
- Tokens may have associated values : the firing of transitions can depend on these values.

Many tools are available to support the development of real-time systems using Petri nets, including editors, tools for analysing system properties, and, often, tools that help drive the system implementation through the lower level phases of the life cycle. Among the available tools are Artifex, Cabernet and Design/CPN. Some have already been applied successfully in industrial projects. For instance, Artifex has been used for about a decade by several Italian and other European companies, largely in automobile manufacturing.

The significant properties of Petri nets are either intractable or undecidable and this is an unfortunate difficulty in modelling with Petri nets. Therefore, simulation is used, in most cases, for analysing properties of interest. An important exception is the Berthomieu-Diaz algorithm [15] for analysing a timed Petri net for reachability. The reachability problem for Petri nets is to determine whether a marking $m'$ can be reached from another marking $m$ through a suitable firing sequence. In some sense, this is the fundamental problem for Petri nets, since many other problems can be reduced to it. Under some reasonably general conditions, this analysis can be completed in polynomial time.

### 3.1.11 GIL

Graphical Interval Logic (GIL) is a visual temporal logic in which formulas resemble the informal timing diagrams used by system designers. It has a formal model-theoretic semantics and can express all properties that can be expressed using linear temporal logic with the *until* operator [41]. It is a linear-time temporal logic that models a computation as a linear sequence of states. The logic allows the user to construct arbitrary intervals of time and to express properties that apply to those intervals. It helps to visualise the relative temporal ordering of states in the system with the horizontal dimension used to indicate the passage of time. The interval operator limits the scope of properties to the interval on which they must hold. The vertical dimension shows the composition of formulas using logical connectives and interval operators. It allows intervals to be nested arbitrarily deeply to express complex temporal relations.

The syntax and semantics of GIL is explained below with some example formulas used to specify the operation of a simple system. The system consists of two concurrent processes that requests the exclusive use of a shared resource. In Figure 3.7, sig1/sig 2 is used to indicate that process 1/process 2 signals for exclusive access to the shared

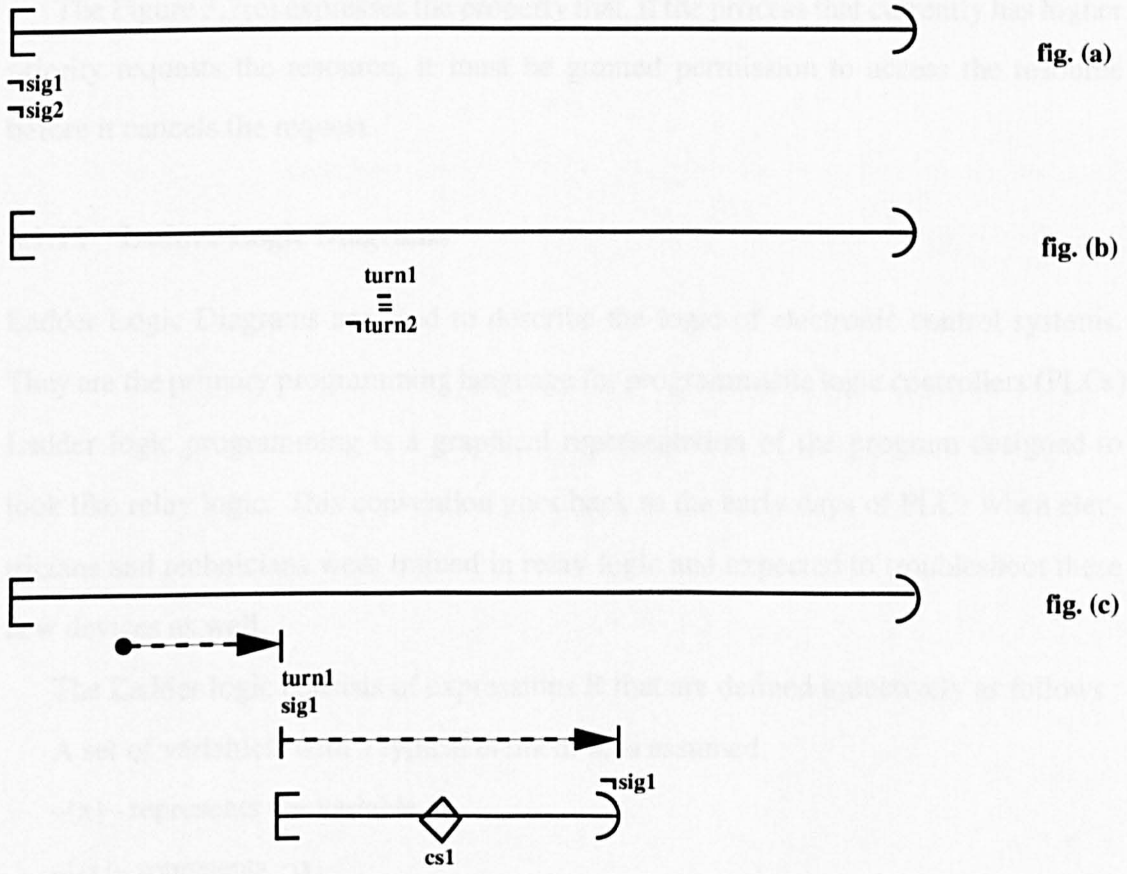


Figure 3.7: GIL Specifications

resource. The formula cs1/cs2 denotes that process 1/process 2 has exclusive permission to use the resource, i.e., it may enter the critical section. The formula turn 1/turn 2 denotes that process 1/process 2 has higher priority than process 2/process 1.

The Figure 3.7(a) asserts that sig1 and sig2 are false at the beginning of the operation. The interval symbol here denotes the entire computation and the formula placed left-justified below it is deemed to hold at the first state of the interval. The formula is formed by the vertical concatenation of two other formulas which indicates their logical conjunction.

The Figure 3.7(b) illustrates how a property can be asserted to be an invariant over an interval. The formula is placed below the interval and indented to the right to indicate that it holds at every state within the interval. Here the formula states that if process 1 has priority, then process 2 does not, and vice versa.

The Figure 3.7(c) expresses the property that, if the process that currently has higher priority requests the resource, it must be granted permission to access the resource before it cancels the request.

### 3.1.12 Ladder Logic Diagrams

Ladder Logic Diagrams are used to describe the logic of electronic control systems. They are the primary programming language for programmable logic controllers (PLCs). Ladder logic programming is a graphical representation of the program designed to look like relay logic. This convention goes back to the early days of PLCs when electricians and technicians were trained in relay logic and expected to troubleshoot these new devices as well.

The Ladder logic consists of expressions R that are defined inductively as follows :

A set of variables, with a typical element x, is assumed.

–[x]– represents the variable x.

–[/x]– represents $\neg$x.

R–[x]– represents the conjunction R $\wedge$ x.

R–[/x]– represents the conjunction $R \wedge \neg x$.

$R_1$ –+– $R_2$ – represents the disjunction $R_1 \vee R_2$.

Since each formula is equivalent to a formula in disjunctive normal form, it is easy to see that each formula can be described by means of an expression R.

A ladder logic diagram consists of a vertical list of assignments of the form

$R_1 : x_1\ R_2 : x_2\ ::\ R_n : x_n$ meaning that the variable $x_i$, called a coil, is assigned the truth value of $R_i$ for i = 1,..,n. Each expression $R_i : x_i$ is called a rung.

A ladder logic diagram consists of 5 types of variables :

- Input Variable : Its value is determined by the environment (i.e., the logistic layer and the infrastructure).
- Output variable : Its value is computed by means of the ladder logic diagram, and passed on to the environment.
- Latch :

  Its value is computed by means of the ladder logic diagram, and not passed on to the environment, but only used in the computation of values of other variables.
- Timer :

  This variable is either on or off, which is determined by the value of its trigger. If it is off, then its value is 0. If it is on, then its value is increased by one with every cycle, until it reaches a preset duration, after which its trigger is switched off (i.e., is assigned the value 0).
- Trigger :

  This indicates whether a certain timer operation is on or off.

Input variables, output variables, latches and triggers have Boolean values (ie., 0 or 1), while the values of timers are natural numbers. Each output variable, latch or trigger $x$ is the coil of exactly one assignment $R : x$ in the ladder logic diagram. Input variables and timers are not allowed as coils. Only input variables, latches and triggers are allowed to occur in the left-hand side of rungs.

[51] reports work related to the conversion of a ladder logic diagram into a Boolean formula, so that the validity of these dependencies in the control tables can be verified using a theorem prover.

### 3.1.13 Z Schema

Z [86] is a typed formal specification language based on set theory and first order predicate logic. It has been developed at Oxford University since the late 1970's by members of the Programming Research Group (PRG). The problems with large specifications using set theory and logic is that specifications become unreadable and unmanageable. Therefore, a schema notation was introduced to aid the structuring of specifications in Z. This provides the framework for textual combinations of sections of mathematics known as schema using schema operators.

A Z schema[1] is presented graphically to highlight its contents within a specification. It normally has two areas: a signature part and a predicate part as shown in Figure 3.8. The signature part contains a declaration of the variables to be used in the schema. The predicate part shows relationships between the variables declared in the signature. The signature part is above the middle dividing line. The predicate part may be omitted in which case there will be no middle line.

The example in Figure 3.8 denotes a Z schema named "S" which introduces a variable of type "A" that can only take values satisfying the predicate "P".

### 3.1.14 Specification Diagrams in UML

The Unified Modelling Language (UML) [140] is a language that unifies the industry's best engineering practices for modelling systems. It is a language, not simply a notation, for capturing knowledge about a subject and expressing knowledge regarding the

[1]"Schema" is not in the plural sense of a scheme ; it is called a "schema" by convention

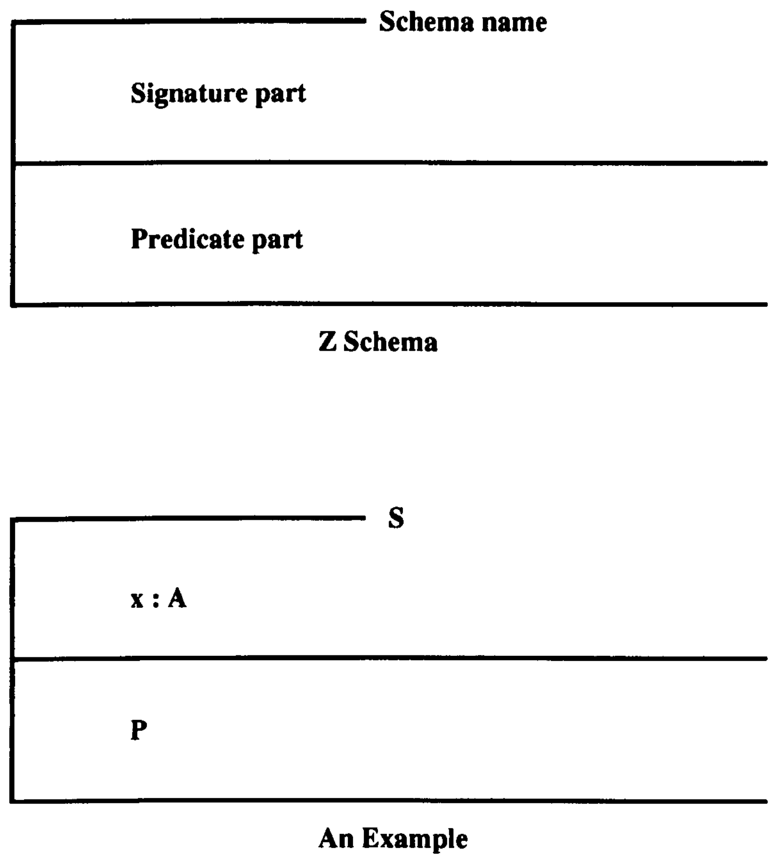


Figure 3.8: A Z Schema

subject for the purpose of communication. It is used for specifying, visualising, constructing, and documenting systems. Figure 3.9 shows the scope of UML. Its goals are to :-

- Be a ready-to-use expressive visual modelling language that is simple and extensible.
- Have extensibility and specialisation mechanisms for extending, rather than modifying, core concepts.
- Allow adding new concepts and notations beyond the core.
- Allow variant interpretations of existing concepts when there is no clear consensus.

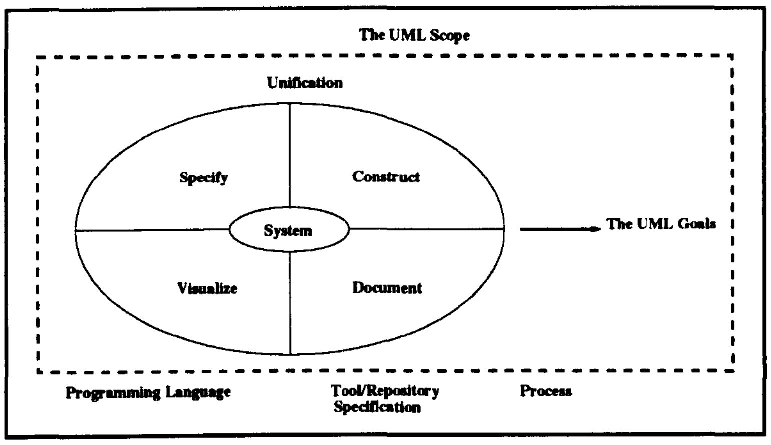


Figure 3.9: The Scope of UML

- Allow specialisation of concepts, notations, and constraints for particular domains.
- Be implementation independent.
- Be process independent.
- Encourage the growth of the object-oriented tools market.
- Support higher-level concepts (collaborations, frameworks, patterns and components).
- Address recurring architectural complexity problems (physical distribution and distributed systems, concurrency and concurrent systems, replication, security, load balancing, and fault tolerance) using component technology, visual programming, patterns, and frameworks.
- Be scalable.
- Be widely acceptable (general purpose and powerful) and usable (simple, widely accepted, and evolutionary).
- Integrate the best engineering practices.

The UML defines nine types of diagrams which are class, object, use case, sequence, collaboration, statechart, activity, component and deployment diagrams.

All of these diagrams are based on the principle that concepts are depicted as symbols and relationships among concepts are depicted as paths (lines) connecting symbols, where both of these types of elements may be named.

Additional information on these diagrams can be obtained in [140].

In summary, one of the important reasons for UML's widespread acceptance is that it has placed a lot of emphasis on diagrams for modelling systems.

## 3.2 Important Design Features for Visual Representations

Based on a study of visual representations in various areas, the following were identified as key features that should be taken into account while designing a visual language.

- Simplicity

  This is one of the most important features of a good visual notation. The simpler the diagram, the easier and quicker it is to extract the meaning from it.

- Intuitiveness

  The diagram should draw the attention of the reader quickly to the suggested meaning.

- Unambiguity

  The diagram should not be causing any confusion in interpretation.

- Readability

  The diagram should have text put at suitable locations to enhance readability without overcrowding the diagram.

- Communicativeness

It should be possible to communicate with other people using the diagram. In other words, it should be possible to suitably abstract the diagram, when necessary, and see the new diagram clearly. It should be possible to navigate the diagram contents with ease.

- Manipulatability

  It should be possible to manipulate the diagram to suitable equivalent forms. The meaning of any manipulation should be clear.

- Composability

  It should be possible to suitably compose two diagrams.

- Customisability

  It should be possible to customise the basic notation in some restricted ways for which guidelines should be provided.

- Realisability

  It should be possible to realize the visual language conveniently in a suitable tool.

- Scalability

  It should be possible to deal with huge descriptions nearly as conveniently as smaller ones.

- Expressivity

  It should have enough expressivity in terms of available language constructs so that expressing anything in context can be done directly rather than by roundabout methods.

## 3.3 Chapter Summary

We have seen how visualisation helps in various domains in fostering increased accessibility of information, languages and technology through various means including better communication and ease of use. We noted that the use of a user-friendly graphical environment facilitated the interaction with and use of computer systems. A wide spectrum of users can be allowed accessibility with the use of such powerful visual environments. Such convenience to the user, particularly in terms of hiding the user from knowing intricate details of textual commands, helps in wider acceptance of such systems. We saw how computer simulations are valuable as research tools for modelling physical system properties that are too difficult, dangerous or expensive to measure. We saw that the success of the visual paradigm for operating systems also encouraged its application to programming languages. Pict/D [58] is an example of a graphical programming environment where icons are used for visual programming. We also examined specification diagrams like Structured Analysis/Real Time (SA/RT), State Transition Diagrams, Statecharts, Statechart-like notations such as Modechart and Timed-Transition Models, Petri Nets, Graphical Interval Logic (GIL), Ladder Logic Diagrams and UML Diagrams. We explained in section 3.2 the important design features to be taken into account while designing a visual language. The application of visualisation aids to an ITL-based formal method will be examined. Before that, the following chapter introduces ITL in detail and highlights some of the difficulties that have to be tackled to increase its uptake by a wider spectrum of users. Having done that, it details the design of a visual notation for ITL and gives examples.

# Chapter 4

# VisITL

The main objective of this chapter is to design a visual notation for the chosen logic-based formalism i.e., ITL [116, 29], based on the design principles identified in the previous chapter. Therefore, this chapter begins with the syntax and semantics of ITL followed by some examples. The current limitations of ITL are also discussed. In a following section, the executable subset of ITL i.e, Tempura, is discussed. After providing this background, a visual notation for ITL is designed and discussed with some examples.

## 4.1 Interval Temporal Logic (ITL)

An interval $\sigma$ is defined as a (in)finite sequence of states $\sigma_0$, $\sigma_1$, $\sigma_2$, .. where $\sigma_i$ is a mapping from the set of variables 'Var' to the set of values 'Val'. The length of $\sigma$ is one less than the number of states in the interval.

In ITL, there are the conventional logical operators such as $\wedge$ and $\neg$ and the predicates. There are temporal operators like $\bigcirc$ and $\Box$ extending the conventional logic to temporal reasoning. Additionally, in ITL, there are temporal operators like ";", "$*$" and "*skip*".

## 4.2 Syntax of the Logic

The syntax of ITL is defined in fig.4.1 where $\mu$ is an integer value, $a$ is a static variable (doesn't change within an interval), $A$ is a state variable (can change within an interval), $v$ is a static or state variable, $g$ is a function symbol and $p$ is a predicate symbol.

| | | |
|---|---|---|
| *Expressions* | $e ::=$ | $\mu \mid a \mid A \mid g(e_1,\ldots,e_n) \mid \imath a\colon f$ |
| *Formulae* | $f ::=$ | $p(e_1,\ldots,e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2 \mid f^*$ |

Figure 4.1: Syntax of ITL

Details of the syntax are explained with some examples below and in section 4.2.2.

1. **Syntax of expressions**

   Expressions are built inductively as follows :

   - Constants : $\mu$

     Constants are denoted by letters of the form $\mu$.

     Examples : $\mu_0$, $\mu_1$ etc. to denote values like 0, 3, and so on.

   - Individual variables : $A$, $B$, $C$, ..,$a$, $b$, $c$, .., $v$,..

     By convention, capital letters are used to denote state variables which are variables whose values can change within an interval and small letters to denote static variables which are variables whose values do not change within an interval. Letters of the form $v$ are used to denote a variable which can either be a static or a state variable.

   - Functions : $g(e_0, e_1, e_2, .., e_k)$ where $k \geq 0$ and $e_0, e_1, e_2, .., e_k$ are expressions. + and mod are among common functions used. Constants (such as 0, 1 etc.) are treated as zero-place functions.

     Examples include : $A + B$, $a - b$, $A + a$, $v \; mod \; C$ and so on.

   - $\imath a\colon f$ : choose a value of $a$ such that $f$ holds. If there is no such $a$, then $\imath a\colon f$ takes an arbitrary value from $a$'s range.

An example of the usage of such an expression is given below :

$\bigcirc$ exp = $\iota a$: $\bigcirc$ (exp = a)

Some examples of syntactically legal expressions are given below:

- $I + (\bigcirc J) + 2$

  This expression adds the value of $I$ in the current state, the value of $J$ in the next state and the constant "2".

- $I + (\bigcirc J) - \bigcirc\bigcirc(I)$

  This expression adds the value of $I$ in the current state to the value of $J$ in the next state and subtracts the value of $I$ in the next to next state from the result.

2. **Syntax of formulae**

   Formulas are built inductively as follows :

   - Predicates : $p(e_0, e_1, e_2, .., e_k)$, where $k \geq 0$ and $e_0, e_1, e_2, .., e_k$ are expressions. For example, $\geq$ is a predicate.

     Examples include : $e_0 \ \geq \ e_7$, $e_3 < e_0$ and so on.

   - Logical connectives : $\neg f$ and $f_1 \wedge f_2$, where $f, f_1$ and $f_2$ are formulas.
   - Universal Quantifier : $\forall v$ . f
   - skip means a unit interval i.e., an interval of length 1
   - chop : $f_1 ; f_2$ holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that $f_1$ holds over the prefix and $f_2$ over the suffix, or if the interval is infinite and $f_1$ holds for that interval.
   - chop-star : $f^*$ holds if the interval is decomposable into a finite number of intervals such that for each of them $f$ holds, or if the interval is infinite and can be decomposed into an infinite number of finite intervals for which $f$ holds.

Some examples of syntactically legal formulas are given below :

- $(J = 2) \wedge \bigcirc(K = 4)$

  This formula states that the value of $J$ is "2" in the current states and the value of $K$ is "4" in the next state.

- $\bigcirc(\Box[I = 2] \wedge \bigcirc\bigcirc[J = 2])$

  This formula states what formula will be true in the next state ; from the next state, $I$ would always be equal to "2" and the value of $J$ in the next to next state would be "2".

Apart from the basic syntax, additional operators are defined in terms of the basic ones as abbreviations. Frequently used abbreviations are listed in tables 4.1–4.4.

Table 4.1: Frequently used Non-temporal Abbreviations

| | | |
|---|---|---|
| $\mathit{true}$ | $\hat{=} 0 = 0$ | true value |
| $\mathit{false}$ | $\hat{=} \neg \mathit{true}$ | false value |
| $f_1 \vee f_2$ | $\hat{=} \neg(\neg f_1 \wedge \neg f_2)$ | or |
| $f_1 \supset f_2$ | $\hat{=} \neg f_1 \vee f_2$ | implies |
| $f_1 \equiv f_2$ | $\hat{=} (f_1 \supset f_2) \wedge (f_2 \supset f_1)$ | equivalent |
| $\exists v \bullet f$ | $\hat{=} \neg \forall v \bullet \neg f$ | exists |

Table 4.2: Frequently used Temporal Abbreviations

| | | |
|---|---|---|
| $\bigcirc f$ | $\hat{=} \mathsf{skip} ; f$ | next |
| $\mathit{more}$ | $\hat{=} \bigcirc \mathit{true}$ | non-empty interval |
| $\mathsf{empty}$ | $\hat{=} \neg \mathit{more}$ | empty interval |
| $\mathit{inf}$ | $\hat{=} \mathit{true} ; \mathit{false}$ | infinite interval |
| $\mathit{finite}$ | $\hat{=} \neg \mathit{inf}$ | finite interval |
| $\Diamond f$ | $\hat{=} \mathit{finite} ; f$ | sometimes |
| $\Box f$ | $\hat{=} \neg \Diamond \neg f$ | always |
| Ⓦ$f$ | $\hat{=} \neg \bigcirc \neg f$ | weak next |
| ⟐$f$ | $\hat{=} \mathit{finite} ; f ; \mathit{true}$ | some subinterval |
| ⊡$f$ | $\hat{=} \neg$(⟐$\neg f$) | all subintervals |

Table 4.3: Frequently used Concrete Abbreviations

| | | |
|---|---|---|
| if $f_0$ then $f_1$ | $\widehat{=}$ if $f_0$ then $f_1$ else empty | if then |
| $fin\ f$ | $\widehat{=} \Box(\text{empty} \supset f)$ | final state |
| $halt\ f$ | $\widehat{=} \Box(\text{empty} \equiv f)$ | terminate interval when |
| $keep\ f$ | $\widehat{=} \boxdot(\text{skip} \supset f)$ | all unit subintervals |
| while $f_0$ do $f_1$ | $\widehat{=} (f_0 \wedge f_1)^* \wedge fin\ \neg f_0$ | while loop |
| repeat $f_0$ until $f_1$ | $\widehat{=} f_0$ ; (while $\neg f_1$ do $f_0$) | repeat loop |

Table 4.4: Frequently used Abbreviations related to Expressions

| | | |
|---|---|---|
| $\bigcirc exp$ | $\widehat{=} \iota a{:}\bigcirc(exp = a)$ | next value |
| $fin\ exp$ | $\widehat{=} \iota a{:}fin\,(exp = a)$ | end value |
| $A := exp$ | $\widehat{=} \bigcirc A = exp$ | assignment |
| $exp_1 \leftarrow exp_2$ | $\widehat{=} finite \wedge (fin\ exp_1) = exp_2$ | temporal assignment |
| $exp_1\ gets\ exp_2$ | $\widehat{=} keep\,(exp_1 \leftarrow exp_2)$ | gets |
| stable $exp$ | $\widehat{=} exp\ gets\ exp$ | stability |
| $intlen\,(exp)$ | $\widehat{=} \exists I \bullet (I = 0) \wedge (I\ gets\ I + 1) \wedge I \leftarrow exp$ | interval length |

### 4.2.1 Model

A model is a triple $(D, \Sigma, M)$ containing a data domain $D$, a set of states $\Sigma$ and an interpretation $M$ giving meaning to every function and predicate symbol. For the discussion here, let the data domain $D$ be the set of integers. A state is a function mapping variables to values in $D$. Let $\Sigma$ be the set of all such functions. For a state s in $\Sigma$, let s[A] denote A's value. Each k-place function symbol g has an interpretation $M$[g] which is a function mapping k elements in D to a single value which is written mathematically as : $M[\text{g}] \in (D^k \rightarrow D)$. Interpretations of predicate symbols map to truth values : $M[\text{f}] \in (D^k \rightarrow \{\text{true, false}\})$. It is assumed that $M$ gives standard interpretations to operators such as + and <. The semantics given here keep the interpretations of functions and predicate symbols independent of intervals. They can however be generalised to allow for functions and predicates that take into account the dynamic behaviour of parameters. Using the states in $\Sigma$, *intervals* can be constructed from $\Sigma^+$, the set of all non-empty, (in)finite sequences of states. If $s, t$ and $u$ are states in $\Sigma$, then the following are possible intervals : $< s >, < sttssust >, < uuu >$ . An interval should contain at least one state. The following introduces a basic notation for manipulating intervals.

Let $I$ denote the set of all intervals. For the sake of discussion here, let $I$ be the set $\Sigma^+$. For an interval $\sigma$ in $I$, let $|\sigma|$ denote the length of $\sigma$. By convention, an interval's length is one less than the number of states. The individual states of an interval are denoted by $\sigma_0, \sigma_1, \ldots.., \sigma_{|\sigma|}$. For example, if the variable $A$ has the value 5 in $\sigma$'s final state, then, the following is true : $\sigma_{|\sigma|} = 5$. In the above model, even though time is viewed as being discrete, it provides a sound basis for reasoning about many interesting dynamic phenomena involving time-dependent and functional behaviour. A discrete time view of the world corresponds, often, to our mental model of digital systems and computer programs. In any case, the granularity of time can be made arbitrarily fine.

The following is a small example regarding the model explained above. Let the data domain $D$ be the set of integers. Let us consider an interval $\Sigma$ consisting of states $\sigma_0$, $\sigma_1$, and $\sigma_2$. Let $a$ be some static variable i.e, variable whose value cannot change within an interval. Let $A$ be some state variable. Then, we give the following interpretation :

$\sigma_0(A) = Value_0$,

$\sigma_1(A) = Value_1$ and

$\sigma_2(A) = Value_2$

where $Value_0$, $Value_1$ and $Value_1$ are the values in the data domain $D$ for the state variable $A$. For the static variable $a$, since its value in the data domain has to be the same throughout the interval, we give the following interpretation :

$\sigma_0(a) = val_a$,

$\sigma_1(a) = val_a$ and

$\sigma_2(a) = val_a$

where $val_a$ is the value in the data domain $D$ for the static variable $a$. In other words, the value of $a$ is the same throughout the interval $\Sigma$.

### 4.2.2 Basic ITL Terminology

The basic terminology of ITL is introduced with the help of examples.

**Operators**

**chop :** The operator ";" is called *chop*. A formula $f_1;f_2$ is true on an interval $\sigma$ iff[1] there is at least one way to split $\sigma$ into two subintervals such that the formula $f_1$ is true on the left subinterval and the formula $f_2$ is true on the right subinterval. That is, $M_\sigma[f_1;f_2]$ = true iff for some $i \leq |\sigma|$, $M_{<\sigma_0,\ldots,\sigma_i>}[f_1]$ = true and $M_{<\sigma_i,\ldots,\sigma_{|\sigma|}>}[f_2]$ = true. It can be noted here that $\sigma_i$ is a common state for the left and right subintervals.

In case the interval is infinite, then, $f_1$ has to be true on $\sigma$.

**empty and more :** The formula *empty* is true on an interval iff the interval has length zero. The formula *more* is just the opposite being true on an interval iff the interval has nonzero length.

**next and weak next :** In order for the construct next viz., $\bigcirc$ to be true on an interval $\sigma$, the length of $\sigma$ must be at least one. Therefore, this operator is referred to as *strong next*. The related construct weak next viz., Ⓦis true on an interval $\sigma$ if either $\sigma$ has length zero or the subformula $w$ is true on $< \sigma_1..\sigma_{|\sigma|} >$. So, we can express the weak next in terms of strong next as : Ⓦ$f \equiv empty \vee \bigcirc f$ where (*next* $w$) means (*strong next* $w$), by default. This way, the weak next provides a concise and natural way to express a construct as a conjunction of its immediate effect and future effect.

**The operator** $\Diamond$ The construct $\Diamond f$ is true on an interval $\sigma$ if there is some suffix subinterval on which the formula $f$ is true :

$M_\sigma[\Diamond f]$ = true iff for some $i \leq |\sigma|, M_{<\sigma_i,\ldots,\sigma_{|\sigma|}>}[f] = true$ and $i \neq \infty$.

In other words, it means that $f$ is true *sometime* in the interval. It can be defined in terms of the chop operator as : $\Diamond f \equiv_{def} finite ; f$

**The operators** ***halt*** **and** ***fin*** This operator can be used, in the form *halt* $f$, to specify that a formula $f$ becomes true only at the end of an interval.

[1]iff is the usual abbreviation for "if and only if" used in mathematics

$halt\ f \equiv \Box(f \equiv empty)$.

For example, $halt\ (I > 10)$ is true on $\sigma$ iff the value of the variable $I$ exceeds 10 in exactly the last state of $\sigma$.

The formula $fin\ f$ is true on an interval $\sigma$ iff the formula $f$ is itself true on the final subinterval $\langle \sigma_{|\sigma|} \rangle$ i.e., the last state of $\sigma$.

$fin\ f \equiv_{def} \Box(empty \supset f)$.

**Temporal equality** The construct $e_1 \approx e_2$ is called temporal equality. It is true iff the expressions $e_1$ and $e_2$ are always equal :

$e_1 \approx e_2 \equiv_{def} \Box(e_1 = e_2)$.

**Length** The formula $len(e)$ is true on an interval having length exactly e.

$M_\sigma[len(e)]$ = true iff $M_\sigma[e] = |\sigma|$.

The length of $\sigma$ can also be denoted as $|\sigma|$.

**Existential quantifier :** Let us consider the states viz., s, t and u, and the values of variables 'M' and 'N' in those states as in table 4.5.

| | M | N |
|---|---|---|
| s | 2 | 4 |
| t | 0 | 4 |
| u | 2 | 3 |

Table 4.5: Example for Existential Quantification

Now, the formula $(\exists I) \centerdot \Box(N = 2 * I)$ is intuitively true on any interval on which we can construct an 'I' such that 'N' always equals twice of 'I' and the length of the interval is the same as that to which the formula corresponds which is 2 here (because there are 3 states under consideration here). In other words, 'N' should always be even on such an interval of length 2. The interval $< ttt >$ satisfies the formula. So does $< sss >$. There are many such intervals which satisfy the formula. $< u >$, $< stut >$ and $< uuu >$ are some intervals that do not satisfy the formula.

**Bounded universal/existential quantifier** Formulas of the following form are referred to as *bounded universal quantification* :

$\forall v \leq e \centerdot f$, where $v$ is a static variable, $e$ is an expression and $f$ is a formula.

$\forall v \leq e \centerdot f \equiv_{def} \forall v \centerdot v \leq e \supset f$.

Formulas of the following form are referred to as *bounded existential quantification*:

$\exists v \leq e \centerdot f$, where $v$ is a static variable, $e$ is an expression and $f$ is a formula.

$\exists v \leq e \centerdot f \equiv_{def} \exists v \centerdot v \leq e \wedge f$.

Of course, the ranges in the above formulas could be $v < e$, $e_1 < v \leq e_2$ etc..

**assignment and gets :** An *assignment* simply gives the value of the expression in the next state. For example, $e_1 := e_2$ means that the value of $e_1$ in the next state will be $e_2$. To say that one expression, say $e_1$, equals another expression, say $e_2$ with a one-unit time delay, the *gets* construct is used to say, $e_1$ *gets* $e_2$. So, in the whole of the interval, at any state, if $e_1$ is true, then $e_2$ will be true in the next state, if there is one.

**Temporal assignment** The formula $e_2 \leftarrow e_1$ is true for an interval if the initial value of the expression $e_1$ equals the final value of the expression $e_2$.

We define this as follows :

$e_2 \leftarrow e_1 \equiv_{def} \exists a : [(a = e_1) \wedge fin(e_2 = a) \wedge finite]$

### 4.2.3 Interpretation of Expressions and Formulas

The interpretation $M$ can be extended, as follows, to give meaning to expressions and formulas on intervals. The constructs $M_\sigma[e]$ and $M_\sigma[f]$ are defined to be equal to the value, in $D$, of the expression e and the formula f, respectively, on the interval $\sigma$.

Let $\chi$ be a choice function which maps any non-empty set to some element in the set. We write $\sigma \sim_v \sigma'$ if the intervals $\sigma$ and $\sigma'$ are identical with the possible exception of their mappings for the variable $v$.

The following list defines some basic operators.

- $M_\sigma[v] = \sigma_0[v]$ , where $v$ is a variable

This means that a variable's value on an interval is its value in the interval's initial state.

- $M_\sigma[g(e_1,..,e_k)] = M[g](M_\sigma[e_1],..,M_\sigma[e_k])$.
- $M_\sigma[\iota a\colon f] = \begin{cases} \chi(u) & \text{if } u \neq \{\} \\ \chi(Val_a) & \text{otherwise} \end{cases}$
  where $u = \{\sigma'(a) \mid \sigma \sim_a \sigma' \wedge M_{\sigma'}[f] = \text{true}\}$.
- $M_\sigma[p(e_1,..,e_k)] = M[p](M_\sigma[e_1],..,M_\sigma[e_k])$.
- $M_\sigma[\neg f]$ = true iff $M_\sigma[f] = false$.
- $M_\sigma[f_1 \wedge f_2]$ = true iff $M_\sigma[f_1]$ = true and $M_\sigma[f_2]$ = true.
- $M_\sigma[\forall v \centerdot f]$ = true iff for all $\sigma'$ s.t. $\sigma \sim_v \sigma', M_{\sigma'}[f]$ =true.
- $M_\sigma[skip]$ = true iff $|\sigma| = 1$.
- $M_\sigma[f_1;f_2]$ = true iff
  (exists a k, s.t. $M_{\sigma_0,..,\sigma_k}[f_1]$ = true) and
  (($\sigma$ is infinite and $M_{\sigma_k,..}[f_2]$ = true) or
  ($\sigma$ is finite and $k \leq |\sigma|$ and $M_{\sigma_k,..,\sigma_{|\sigma|}}[f_2]$ = true)))
  or ($\sigma$ is infinite and $M_\sigma[f_1]$).
- $M_\sigma[f^*]$ = true iff
  if $\sigma$ is infinite then
  (exist $l_0,...,l_n$ s.t. $l_0 = 0$ and
  $M_{\sigma_{l_n},..}[f]$ = true and
  for all $0 \leq i < n, l_i < l_{i+1}$ and $M_{\sigma_{l_i},..,\sigma_{l_{i+1}}}[f]$ = true)
  or
  (exist an infinite number of $l_i$ s.t. $l_0 = 0$ and
  for all $0 \leq i, l_i < l_{i+1}$ and $M_{\sigma_{l_i},..,\sigma_{l_{i+1}}}[f]$ = true)
  else

(exist $l_0, \ldots, l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and

for all $0 \leq i < n, l_i < l_{i+1}$ and $M_{\sigma_{l_i}, \ldots, \sigma_{l_{i+1}}}[f]$ = true).

### 4.2.4 Validity and Satisfiability

#### Interpretation of a formula

Let s, t and u be states in which the variables 'I' and 'J' have the values as shown in table 4.6.

| | I | J |
|---|---|---|
| s | 1 | 2 |
| t | 3 | 4 |
| u | 3 | 1 |

Table 4.6: Values of I and J in 3 Different States

The formula (J=2) $\wedge$ $\bigcirc$ (I=3) is true on an interval $\sigma$ iff $|\sigma| \geq 1$, the value of 'J' in the first state i.e., $\sigma_0$ is 2 and the value of 'I' in the next state i.e., $\sigma_1$ is 3. Thus the formula is true on the interval $< stu >$. On the other hand, the formula is false on the interval $< ttu >$ because the initial value of 'J' on this interval is 4 instead of 2.

#### Validity of a formula

Consider the following formula : $\Box(I = 1) \wedge \bigcirc(I = 2)$. This formula is true on an interval $\sigma$ if $|\sigma| \geq 1$, the variable 'I' always equals 1 and in the state $\sigma_1$, 'I' equals 2. No interval can have all of these characteristics. Therefore, the formula is false on all intervals and its negation is always true and hence valid. It is written as follows :

$\models \neg(\Box(I = 1) \wedge \bigcirc(I = 2))$.

The $\models$ stands for "is valid".

### 4.2.5 Proof System for ITL

A proof system is required in any logic so that reasoning can be performed by applying appropriate rules. As an example of a rule in propositional logic, let us consider the simple rule of double negation. It simply states that there is no difference between $\phi$ and $\neg\neg\phi$ which is intuitively true. Applying this rule to the sentence "It is not true that it does **not** rain.", we can infer the sentence "It rains". Books on logic will show several such rules in propositional logic and predicate logic and how they can be applied to infer conclusions from premises. [80] is one such good reference.

Similarly, for ITL, there are several rules in its proof system. Soundness and Completeness are important issues with respect to such a proof system. For example, the soundness of propositional logic is useful in ensuring the non-existence of a proof for a given conclusion from a given set of premises. In other words, soundness will let us know if a formula cannot be inferred from a set of other given formulas. Without knowing this bit of information, a failure in establishing a proof will not let us know whether it is due to our lack of skill or whether such a proof cannot be constructed at all. Completeness of a proof system implies that any true formula in that logic can be proved true using some set of proof rules in the system.

The following table 4.7 gives the propositional rules and axioms for ITL.

Table 4.7: Propositional Axioms and Rules for ITL.

| | | |
|---|---|---|
| ChopAssoc | $\vdash$ | $(f_0;f_1);f_2 \equiv f_0;(f_1;f_2)$ |
| OrChopImp | $\vdash$ | $(f_0 \vee f_1);f_2 \supset (f_0;f_2) \vee (f_1;f_2)$ |
| ChopOrImp | $\vdash$ | $f_0;(f_1 \vee f_2) \supset (f_0;f_1) \vee (f_0;f_2)$ |
| EmptyChop | $\vdash$ | $\mathsf{empty};f_1 \equiv f_1$ |
| ChopEmpty | $\vdash$ | $f_1;\mathsf{empty} \equiv f_1$ |
| NextImpNotNextNot | $\vdash$ | $\bigcirc f_0 \supset \neg\bigcirc\neg f_0$ |
| BoxInduct | $\vdash$ | $f_0 \wedge \Box(f_0 \supset \textcircled{w} f_0) \supset \Box f_0$ |
| InfChop | $\vdash$ | $(f_0 \wedge \mathit{inf});f_1 \equiv (f_0 \wedge \mathit{inf})$ |
| ChopStarEqv | $\vdash$ | $f_0^* \equiv (\mathsf{empty} \vee ((f_0 \wedge \mathit{more});f_0^*))$ |
| MP | $\vdash$ | $f_0 \supset f_1, \vdash f_0 \Rightarrow \vdash f_1$ |
| BoxGen | $\vdash$ | $f_0 \Rightarrow \vdash \Box f_0$ |

Some axioms for the first order case are shown in table 4.8 where $v$ refers to both static and state variables, and $f_v^e$ denotes that in the formula $f$, expression $e$ is substituted for variable $v$. For example, in the SubstAxiom axiom in table 4.8, in formula $f$, the state variable $B$ is substituted for $A$.

[117] and [118] are references for work on completeness of the ITL proof system. The following paragraphs briefly explain some of the above proof rules.

Table 4.8: Some First Order Axioms and Rules for ITL.

| | |
|---|---|
| ForallSub | $\vdash \forall v \cdot f \supset f_v^e$, where the expression $e$ has the same data and temporal type as the variable $v$ and is free for $v$ in $f$. |
| ForallImplies | $\vdash \forall v \cdot (f_1 \supset f_2) \supset (f_1 \supset \forall v \cdot f_2)$, where $v$ doesn't occur freely in $f_1$. |
| SubstAxiom | $\vdash \Box(A = B) \supset f \equiv f_A^B$. |
| StaticWeakNext | $\vdash w \supset \text{Ⓦ}w$, where $w$ only contains static variables. |
| ExistsChopRight | $\vdash \exists v \cdot (f_1;f_2) \supset (\exists v \cdot f_1);f_2$, where $v$ doesn't occur freely in $f_2$. |
| ExistsChopLeft | $\vdash \exists v \cdot (f_1;f_2) \supset f_1;(\exists v \cdot f_2)$, where $v$ doesn't occur freely in $f_1$. |
| ExistsImpDesc | $\vdash (\exists v \cdot f) \wedge (\iota v{:}f) = v \supset f$, where $v$ is a static variable. |
| ForallGen | $\vdash f \Rightarrow \vdash \forall v \cdot f$, for any variable $v$. |

The rule "ChopAssoc" states that the chop operator of ITL is associative just as the name of the rule implies. So, for example, if you can "chop" an interval such that $f_0$ ; $f_1$ is true in the left subinterval and $f_2$ is true in the right subinterval, then, you can also chop the same interval into a left subinterval where $f_0$ is true and a right subinterval where $f_1$ ; $f_2$ is true. In other words, the chop operator is associative.

The "ChopEmpty" is a rule stating that any formula followed by the chop operator and empty is equivalent to the formula itself. This simply follows from the definition of "chop" and "empty" operators.

The "BoxInduct" states if $f_0$ is true in the initial state and always $f_0$ being true implies the formula $\text{Ⓦ}f_0$, then, always $f_0$ is true can be inferred.

The "InfChop" rule simply states that if $f_0$ is true and the interval is infinite in the left subinterval of a "chop", then, the right hand side formula is immaterial.

The definitions of the ITL constructs can be used to understand the above axioms and rules for ITL. It is important to note that there is a proof system for ITL that can help in performing deductive reasoning in ITL.

The importance of these rules lie in the fact that required, important properties of an

ITL specification can be formulated in ITL themselves and then proved to be satisfied by the specification (or otherwise) using the proof system for ITL. At present, the proof system of ITL has been embedded in the Prototype Verification System (PVS) proof tool [84] to enable computer-assisted proof construction. It is worth noting here that a lot of guidance is needed to perform proof tasks and hence only experts in theorem proving who are experienced in such tasks can do it. Further work in this area will involve the creation of proof strategies to assist non-experts in performing ITL proofs. With such proof strategies being offered to inexperienced users, the users would be able to select a proof strategy and let the computer automatically perform the proof tasks involved in sequence.

### 4.2.6 Examples using the ITL Constructs

1. **Tree summation [116]**

   Let us consider a binary tree such as either of the ones shown in the linearly represented list structures :
   $\langle\langle\langle 1,2\rangle,\langle 3,4\rangle\rangle,\langle\langle 5,6\rangle,\langle 7,8\rangle\rangle\rangle$ and $\langle\langle 1,\langle 2,3\rangle\rangle,\langle 4,5\rangle\rangle$.

   Let the function $leaf\_sum(tree)$ determine the sum of a tree's leaves :

   $leaf\_sum(tree)\ \widehat{=}$
   $if\ is\_integer(tree)\ then\ tree$
   $else\ leaf\_sum(tree_0) + leaf\_sum(tree_1)$.

   The predicate $is\_integer$ is true when the parameter $tree$ is an integer(i.e., a leaf) and $false$ when $tree$ is a pair.

   Let us now consider the task of designing an algorithm to reduce a tree-in-place to a single value indicated by $leaf\_sum$. If the variable $Tree$ initially equals such a binary tree, we can specify the problem as :
   Tree $\leftarrow leaf\_sum(Tree)$.

   The following are predicates for a serial solution :

```
$serial\_sum\_tree(Tree) \mathrel{\widehat{=}}$
$if\ is\_integer(Tree)\ then\ empty$
$else($
$sum\_subtree(Tree,0);$
$sum\_subtree(Tree,1);$
$($
$skip \wedge (Tree \leftarrow Tree_0 + Tree_1)$
$)$
$).$
```

```
$sum\_subtree(Tree,i) \mathrel{\widehat{=}}$
$serial\_sum\_tree(Tree_i)$
$\wedge\ stable\ Tree_{1-i}.$
```

The following may be noted :

- If the tree is already an integer-valued leaf, *serial_sum_tree* terminates. Otherwise, the predicate *sum_subtree* is used to reduce the left subtree first and then the right subtree.
- $skip \wedge (Tree \leftarrow Tree_0 + Tree_1)$ is used to finally reduce the tree to a single value.
- The tree is initialised and *serial_sum_tree* is invoked by a formula of the following form :
  $Tree = \langle\rangle$
  $\wedge\ serial\_sum\_tree(Tree) \wedge \Box display(Tree).$

The following are predicates for a parallel solution :

```
$par\_sum\_tree(Tree) \mathrel{\hat{=}}$
$if\ is\_integer(Tree)\ then\ empty$
$else($
$\exists Done : ($
$(Done \approx [is\_integer(Tree_0) \wedge is\_integer(Tree_1)])$
$\wedge\ halt\ Done$
$\wedge\ sum\_sub\_tree\_process(Done, Tree_0)$
$\wedge\ sum\_tree\_process(Done, Tree_1)$
$);$
$($
$skip \wedge (Tree \leftarrow Tree_0 + Tree_1)$
$)$
$).$
```

```
$sum\_tree\_process(Done, Tree) \mathrel{\hat{=}}$
$process($
$par\_sum\_tree(Tree);$
$($
$halt\ Done$
$\wedge\ stable\ Tree$
$)$
$).$
```

The following may be noted :

- The predicate *par_sum_tree* is similar to *serial_sum_tree* except that it recursively reduces each half of a pair in parallel rather than sequentially.
- The variable *Done* is used to monitor the progress of the two subtrees. It equals true when they are both finally integers.

- The subordinate predicate *sum_tree_process(Done, Tree)* reduces its tree parameter and keeps the Tree stable until the flag *Done* becomes true. This ensures that the two parallel invocations of *sum_tree_process* finish at the same time.

2. **An example of an automobile cruise control system**

Let us consider an **automobile cruise control system**. A cruise control system sets the speed of a vehicle with the touch of a finger. Once the system is set, unintended speeding is avoided. The system constantly measures the changes in engine loading and vehicle speed. Functions on the steering wheel allow for slowing down or accelerating without touching the accelerator (gas pedal). So, a cruise control system increases comfort by reducing fatigue and prevents fuel wastage from unintended speeding.

The state space of the system is represented by the following variables and their constraints :

- **Driver Input**

  Driver initiated inputs are represented by **EngineState, BrakeState, GasState** and **CruiseVal**. The possible values for the variables are given by the sets below :

  **EngineState** = {off, on}

  **BrakeState** = {off, on}

  **GasState** = {off, on}

  **CruiseVal** = {constant, neutral, off, resume}

- **Car Speed**

  Information on car speed is represented by **SpeedVal** and **SpeedState**. Additionally, **DesiredSpeedSetting** can be set to any real value from Zero to Max, indicating the speed to be attained by the cruise control system.

**SpeedVal** = {Zero..Max}

**SpeedState** = {Const, Up, Down}

**DesiredSpeedSetting** = {Zero..Max}

- **Car Internal States**

  The internal state of the car is modelled by the state variable **CarInternal-State**. It takes values from the following set :

  **CarInternalState** = {Idle, Manual, Accelerating, Cruising, Resuming}

We can now specify a constraint on the state space as below :

(a) **If the value of the speed is either Zero or Max., then, SpeedState is going to be constant**

Implicit in this constraint is the fact that the **SpeedState** may or may not be constant when the **SpeedVal** is neither Zero or Max.

Its ITL specification is :

$\mathbf{SpeedVal} = \mathit{Zero} \vee \mathbf{SpeedVal} = \mathit{Max} \supset \mathbf{SpeedState} = \mathit{Const}$

**The Initial State Specification**

We can define a formula, **FInitial**, for initial states of the cruise control system. The initial states include, say, the engine being off, the cruise lever set to off, the car speed being zero, the internal state being idle and the desired speed not being set yet.

Its ITL specification is :

$\mathbf{FInitial} \mathrel{\widehat{=}} \mathbf{CarInternalState} = \mathit{Idle} \wedge \mathbf{CruiseVal} = \mathit{off} \wedge \mathbf{SpeedVal} = \mathit{Zero} \wedge$
$\mathbf{EngineState} = \mathit{on}$

**Safety and Liveness Specification**

**If the engine is off, then, the car will eventually stop**

Turning off the engine implies that eventually the speed will become zero.

Its ITL specification is :

$\textbf{EngineState} = off \supset \Diamond\textbf{SpeedVal} = \textbf{Zero}$

**If the lever of cruise const is applied, and the brake is off at that time, the desired speed, DesiredSpeedSetting, will eventually be set, whether it has been set previously or not.**

Its ITL specifications is :

$\textbf{BrakeState} = off \wedge \textbf{CruiseVal} = constant \supset \Diamond\textbf{SpeedVal} = \textbf{DesiredSpeedSetting}$

**Examples with real time constraints :**

Let us first make some assumptions and introduce certain constants. Let *CAcceleration* denote the magnitude of a constant value of both acceleration and deceleration rates. The maximum value for the car speed has already been introduced as *Max*. Hence the maximum time required for a car to achieve its maximum speed, say CMaxTime, will be Max/CAcceleration. Let CInfinity denote an arbitrary large number for time greater than CMaxTime.

Now we can introduce some example specifications with real-time constraints.

**If the current speed is below the desired speed, and if the car increases its speed continuously for a period longer than CMaxTime, then the speed of the car will eventually reach the desired speed.**

Its ITL specification is :

$\textbf{SpeedVal} < \textbf{DesiredSpeedSetting} \wedge \textbf{SpeedState} = \textbf{Up}$
$\supset \textbf{t} \geq CMaxTime \wedge \textbf{t} < CInfinity \wedge \Diamond\textbf{SpeedVal} = \textbf{DesiredSpeedSetting}$

Once again, it can be noted that textual specifications are monotonous in their appearance in that they are full of text and symbols. The structuring of the specifications in intuitive graphical notations will be one big step forward in attempts to increase the accessibility of ITL specifications.

### 4.2.7 Limitations in the usage of ITL

The following explains the limitations in the usage of ITL.

- Readability of the specifications has to be enhanced. In this context, we have already seen the use of abbreviations which were defined in terms of the basic constructs. This is not enough as ITL is a mathematical approach involving many symbols. This is especially a problem when large specifications are considered. Moreover, this hinders wider accessibility.

- In order to address schedulability and resource allocation problems, ITL has to be integrated with another suitable formalism. An approach along the lines of [26], which has an example where the formalism Temporal Agent Model (TAM) [105, 135, 136] was given ITL semantics and used to specify a robot control system, is required.

- In order to prove properties about the system specification in ITL, a user-friendly proof tool has to be integrated with ITL. ITL has been embedded in the Prototype Verification System (PVS) thus enabling ITL proofs to be worked out. However, a model-checker is necessary so that the state space based on an automata representation for an ITL formula can be explored to check for required properties. [117] describes some work in this direction.

- Some benchmark case-studies have to considered to demonstrate the suitability of ITL for the specification of real-time safety critical systems. The following case-studies are good case-studies for such benchmarking : The mine pump control system [91], a robot control system for a robot exhibiting some specified behaviour which is more complicated than in [26] and railroad crossing systems [74]. These case-studies would also demonstrate the scalability potential of ITL.

  Apart from this, new case-studies including new domains have to be considered in order to test these techniques as well as increase confidence in them. This will go a long way in increasing their accessibility to industrial users.

- Software tool support that enables a user to either write specifications in some graphical language or ITL specifications directly and, if necessary, convert to the other form would help ITL become more accessible. This tool would help inexperienced users to specify using the more convenient graphical language and, if interested, learn ITL using this tool. The resulting ITL specification could be linked with other tools for simulation or proving properties. The experienced ITL user could always write ITL specifications directly. Even for such a user, visualising the ITL specification would be possible if the ITL specification is convertible to a graphical form.

- Refinement and proofs performed on ITL specifications will further burden the user in terms of formal methods and hence, if they are performed on intuitive graphical ITL specifications, various levels of users can be accommodated in such processes which are particularly important while developing safety-critical systems.

## 4.3 Execution of Specifications

### 4.3.1 Executable Specifications

Specifications play a central role in systems development in more than one way. They define all the required characteristics of the system to be implemented. To define the required characteristics precisely and concisely, specifications must be written in formal and highly expressive languages [151]. For an immediate reflection of the consequences of the specifications, and for an early validation, it has been suggested that specifications should, furthermore, be executable [2]. Hayes and Jones [70], however, argue that the demands of high expressiveness and executability are mutually exclusive. Norbert E. Fuchs [55] argues that high expressiveness and executability need not exclude each other if specifications are written in declarative languages. The following summarises the arguments for the use of executable specifications written in declarative languages [55].

- Executable specifications allow us to demonstrate the behaviour of a system before it is actually implemented. This has the following positive consequences for systems development.

  - Executable components are available much earlier than in the traditional life-cycle. Therefore, validation errors can be corrected immediately, without incurring costly development.
  - Requirements that are initially unclear can be clarified and completed by hands-on experience with the executable specifications.
  - Execution of the specification supplements inspection and reasoning as means for validation. This is especially important for the validation of non-functional behaviour.

  Declarative languages, especially logic languages, combine high expressiveness with executability. They allow us to write both property-oriented and model-oriented executable specifications on the required level of abstraction. Logic specification languages permit us to express non-determinism in a natural way.

- Executable specifications are constructive i.e., they not only demand the existence of a solution, but also, actually construct it.

- As long as there is an executable subset in the specification language, the non-executable specification can be transformed by the process of refinement (and the incorporation of appropriate design decisions) into an executable specification. This executable specification may either be at almost the same level of abstraction as the non-executable one or at a very different level of abstraction.

- Executable specifications do not constrain the choice of possible implementations because only minimal design and implementation decisions are necessary to obtain executability. In addition, these decisions are revisable.

- Verification of an implementation against the specification becomes superfluous if we use the transformational approach. This is because transformations performed using correctness-preserving sound rules result in verification automatically being done in each of the transformation steps. This is not the case when one starts with a specification and end up with an implementation without any formal transformations. In such a case, verification is necessary to show that the implementation is a correct refinement of the specification.

### Executable Temporal Logics

The development of executable temporal logics is becoming increasingly important while a variety of verification systems based upon temporal logic, particularly involving model-checking techniques, have been produced. An introduction to five temporal logic-based programming languages : Chronolog, F-LIMETTE, Concurrent METATEM, Tempura and Tokio can be found in [49]. These languages can be classified depending on the type of logic upon which they are based, their execution schemes and their applicability. Chronolog and Concurrent METATEM use Linear-Time Temporal Logic (LTTL); Tempura and Tokio use Interval Temporal Logic (ITL); and F-LIMETTE uses Metric Temporal Logic (FMTL). Concurrent METATEM and Tempura use deterministic execution schemes suitable for practical programming languages while others feature backtracking mechanisms. Concurrent METATEM is naturally applicable to concurrent object-based (and agent-based) systems, while the others are intended for single object implementation. Thus, together, these languages cover much of the range of elements being actively explored throughout the field of executable temporal logics.

The following are some useful WWW sites:

- http://www.csl.sri.com/lucid/intense for InTense, a language which includes the original Chronolog as a subset.

- http://www.cse.dmu.ac.uk/~cau/itlhomepage/index.html for Tempura.

### 4.3.2 Introduction to Tempura

Tempura provides an executable framework for developing and experimenting with suitable ITL specifications. It is a programming language based on ITL and, hence, is an executable subset of ITL. In what follows, we discuss Tempura and outline its limitations. For more discussion, we refer the reader to [116, 84].

### 4.3.3 Syntax of Tempura

The syntax of Tempura corresponds to the syntax of the executable subset of ITL. The basic statements of Tempura include the more, empty, $A = exp$, true, false, if then else, while, repeat constructs, the ITL $f_1 \wedge f_2$ represented as $f_1$ *and* $f_2$, $\Diamond$ represented as sometimes and $\Box$ represented as always. Further details can be obtained from [116].

### 4.3.4 Tempura Interpreter

Let us consider the following Tempura program : (((next)(next)empty) and (I=0) and (I gets (I+1)) and always(J=2.I)). This program is simple enough to make a mental calculation and come to the conclusion that this is true on intervals of length 2 in which 'I' assumes the value 0, 1 and 2 while 'J' simultaneously assumes the values 0, 2 and 4. One way to execute such a formula is to transform it to a logically equivalent conjunction of two formulas as (*present_state* and (weak next)*what_remains*) where the formula *present_state* consists of assignments to the program variables and also indicates whether or not the interval is finished while the formula *what_remains* is what is executed in subsequent states if the interval does indeed continue on. For the formula under consideration, *present_state* has the value, ( (I=0) and (J=0) and more). The value of *what_remains* is the formula, (((next)empty) and (I=1) and (I gets(I+1)) and always(J=2.I)). A detailed account of the transformation of the formula is given in Figure 4.2.

The operation of the Tempura interpreter is based on the technique just described. For details on the variables used by the interpreter and the basic execution algorithm,

**Before state 0** : ((next)(next) empty and (I=0) and (I gets (I+1)) and always(J=2.I))

**After state 0** : [(I=0) and (J=0) and more] and
(weak next)[ (next empty) and (I=1) and (I gets (I+1)) and always(J=2.I)]

**Before state 1** : (next empty) and (I=1) and (I gets (I+1)) and always(J=2.I)

**After state 1** : [(I=1) and (J=2) and more] and
(weak next)(empty and (I=2) and (I gets (I+1)) and always(J=2.I)]

**Before state 2** : empty and (I=2) and (I gets (I+1)) and always(J=2.I)

**After state 2** : [(I=2) and (J=4) and empty] and
(weak next)[false and (I gets (I+1)) and always(J=2.I)]

Figure 4.2: Transformation of the Formula by the Interpreter

we refer to [116]. The various constructs of Tempura are implemented by using re-write rules [116].

### 4.3.5 Examples in Tempura

1. The following is a very simple example.

   **Problem description :** There are three system variables, X, Y, and Z. The user inputs the initial values of X and Y. The value of X in subsequent states is to be incremented by a constant value of 10 and the value of Y in subsequent states is to be decremented by a constant value of 1. **Z should take the sum of the values in X and Y.** This should continue for an interval length of n to be specified by the user. When the final state is reached, the square of Z is to be displayed.

   **A sample Tempura program :**

```
define main() = {
exists X,Y,Z,n : {
define first(n) = {
{empty and input X and input Y and input n};
{X gets X+10 and Y gets Y-1 and Z gets X+Y and len(n)} ;
{empty and output X and output Y and output Z*Z}
}
and
define second() ={
always {
format("X = %tY = %tZ = %t/ n",X,Y,Z) }
}
and
second() and first(n)
}
}.
```

**The output :**

```
Tempura 12> load "mono1".
[Reading file /export/home0/users/arunc/tempu_ex/ok/mono1.t]
Tempura 13> run main().
State0 :> X =?10.
State0 :> Y =?100.
State0 :> n =?3.
State0 : X = 10  Y = 100  Z =?
State1 : X = 20  Y = 99  Z = 110
State2 : X = 30  Y = 98  Z = 119
State3 : X = 40
State3 : Y = 97
State3 : (Z * Z) = 16384
State3 : X = 40  Y = 97  Z = 128
```

```
Done! Computation length: 3. Total Passes: 7.
Total Reductions: 175 (150 successful).
Maximum reduction depth: 10.
```

**Comments :** It must be noted that Z is shown undefined in the initial state and is in accordance with the specification. The value of Z in state 1 is 110 as shown and not 119 as one might expect by mistake. If 119 was actually required in the system, then the statement Z = X + Y would have to be used instead of Z gets X + Y. The specification in this case would have **"Z should take the sum of the values in X and Y in the current state"**. This is a simple example where simulation results can help in communication between the customer and the developer at an early stage in the development cycle and get things right.

It can also be noted that the statement *exists* in the program which is used to declare variables is not very good for readability. There is, in fact, a lot of *syntactic additions* to be done in Tempura.

### 4.3.6 Non-executability in Tempura

'I gets (I+1)' is a non-executable statement since the initial value of 'I' is undefined in the above statement. ((I=0) and (I gets (I+1)) has an initial value for 'I' but the execution will not terminate. (next(I=1) or next(I=2)) is non-executable because the value of 'I' in the next state is non-deterministic. The use of 'sometimes' also can cause a problem as can be seen in the following example : ((I=0) and (I gets (I+1)) and halt(I=2) and sometimes(I=3))) where *halt w* means that the formula *w* becomes true only at the end of the interval under consideration. Therefore, the program is supposed to stop execution when 'I' gets the value '2' apart from 'I', in some state, having had the value of '3', which is clearly not going to be the case. So, one has to be cautious while writing programs in Tempura.

Here are some example programs and the corresponding outputs :

- **Tempura program with undefined initial value :**

```
define main() = {
exists X : {
define first() = {
{(X gets X + 1)}
}
and
define second() = {
always{
format("X=%t / n", X)}
}
and
second() and first() }
}.
```

```
Tempura 9> load "nonexec".
[Reading file /export/home0/users/arunc/tempu_ex/ok/nonexec.t]
Tempura 10> run main().
State0 : X =?
***Tempura error : state #0 (pass#3) is not completely defined.
Evaluating: X gets (X + ...)
Undefined: {empty}{}{X}{}{}
Abort (a), Break (b) or Continue (c)?
```

- **Tempura program with Sometimes**

```
define main() = {
exists X : {
define first() = {
```

```
{ (X=0) and (X gets X+1) and halt(X=9) and sometimes(X=10)}
}
and
define second() ={
always {
format("X = %t/ n",X) }
}
and
second() and first()
}
}.
```

```
Tempura 11> load "sometime".
[Reading file /export/home0/users/arunc/tempu_ex/ok/sometime.t]
Tempura 12> run main().
State0 : X = 0
State1 : X = 1
State2 : X = 2
State3 : X = 3
State4 : X = 4
State5 : X = 5
State6 : X = 6
State7 : X = 7
State8 : X = 8
***Tempura error: attempt to re-assign interval length.
Evaluating: next sometimes (X = 10)
Abort (a), Break (b) or Continue (c)?
```

### 4.3.7 Limitations of Tempura

**Increasing Readability of Specifications :**

Constructs could be added in Tempura in order to increase readability. While shared variables could be used for synchronisation between processes, having explicit channel communication constructs in ITL/Tempura would be better. To be able to deal with deadlines for computations or delay, we could use constructs which mean these by virtue of their names and this will lead to more readable specifications.

**Improvements to User-interface :**

There is no doubt that a helpful and "user-friendly" interface is required in order for these software techniques to become popular. If such a user-interface is easy to learn, simple to use, straightforward, and forgiving, the user will be inclined to make good use of these techniques. In order to achieve this, amongst many other things, it would be necessary to insulate the user from the intricacies that are of no consequence to the user and provide only useful information for effective interaction and hence a proper and efficient usage of these software techniques. Therefore validating user inputs, handling errors and displaying appropriate error messages would be very important. Also, features like 'on-line help' and giving useful examples and possibly even allowing the user to customise the interface will all go a long way towards enabling more people to become aware of the usefulness of these techniques.

**Integration with Other Tools/Techniques :**

With the help of Tempura, we can write a specification as has already been discussed. Executing the specification and then experimenting with the specification will give us a lot of insight into the system/program that we hope to develop. In order to have confidence in what we hope to develop, which is especially crucial if the applications are safety-critical, we would need to mathematically prove that certain properties are satisfied by the specification. For this task of proving properties, there are some excellent tools already available. Prototype Verification System (PVS) is such a tool [131]. Though they are excellent from the performance point-of-view, these tools are suited only for experienced users. In our proposed lean approach, we aim to get over these problems and provide better accessibility to proof techniques. Ultimately, we aim to have such an integrated set of tools to be able to do all the various possible things with a specification before actually developing the safety-critical systems. Also, there are possibilities for improving upon the techniques used by Tempura and these have to be examined. For example, Tempura does not keep track of the information in past states and hence it cannot use backtracking techniques. Therefore, it is very deterministic at the moment.

**Automatic Code Generation :**

This would involve converting the final specification into Ada or C code. Of course, the use of the above tools and techniques will help in ensuring that after each step of refinement from the first specification to the final one, the specification continues to satisfy the desired properties. Since the final code in Ada/C would be extracted from a Tempura program about which we would be more confident, it will lead to systems that can be more confidently deployed in safety-critical applications.

A step further would be to either have the compiler from Tempura to Ada/C validated or to have the process of translation from Tempura carried out through some defined correctness-preserving refinement laws.

**A Note on Case Studies :**

These techniques are yet to be tested on industrially-relevant case studies. Such an effort can lead to improvements that can make these techniques popular with industry. Many case studies like those relating to control systems for a robot used for bomb disposal or control systems for a nuclear reactor or timing problems in mobile phone communication systems can be considered to test and demonstrate these techniques.

## 4.4 Summary of the Sections on ITL

In the above sections, Interval Temporal Logic (ITL) has been introduced in detail. Example specifications are given highlighting some of the problems with textual ITL. An introduction to Tempura, an executable subset of ITL, is also given with some examples. The following section details possible visualisation approaches in ITL as a means of increasing the uptake of ITL as a formal method.

## 4.5 VisITL

In chapter 3, we examined ways in which visualisation helps in various domains.

Possible approaches to visualisation in ITL include :

- **Graphical output for simulation through Tempura**

  4.3 introduced Tempura, an executable framework for developing and experimenting with suitable ITL specifications. For such simulations through Tempura, a graphical output is now possible due to the work of Cau [84]. If a graphical output is desired, then, a Tempura file has to be provided with a Tcl/Tk file to define the graphics accordingly. This helps in visualising the simulation output and thus provides an insight into the ITL specification.

- **Automata representation**

  [117] and [118] describe work on representing ITL formulas as an automaton. As we have seen before, a finite automaton can be represented as a state transition diagram which is a directed graph wherein the nodes correspond to the states and the edges correspond to the transition as indicated by the transition function. Briefly, the so-called Chop-automaton for ITL is defined as a quintuple $(V, K, q_0, \delta, \tau)$ for which $V$ is a possibly empty finite set of boolean and numerical

state variables, $K$ is a nonempty finite set of automaton states, $q_0 \in K$ is the initial state, $\delta$ is the transition function mapping $K \times K$ to quantifier-free state formulas over variables in V, $\tau$ is the termination function mapping $K$ to quantifier-free state formulas over variables in $V$.

- **Graphical Notation For ITL**

  This is one interesting possibility with respect to visualisation for ITL which has been previously unexplored and which forms the basis of this work. The following sections detail the design rationale and the visual notation itself. I wish to clarify here that the visual language is designed with the purpose that the user shall only be required to learn the visual language and not be required to know the notations of textual ITL. An implementation would be able to take care of this translation; such an implementation is described in chapter 7.

## 4.6 Design Rationale for a Visual Notation for ITL

In section 3.2, key features of various representations were described and summarised. Let us now look at how the development of a visual notation for ITL commenced.

### 4.6.1 Key Requirements for a Visual Notation for ITL

In the development of a visual language for ITL, a textual, formal logic based language, the following were identified as major requirements and hence were the primary influences on our first visual notations.

- Simplicity
- Intuitiveness
- Unambiguity
- Readability

- Communicativeness
- Manipulatability
- Composability
- Customizability
- Expressivity

### 4.6.2 Experiences in the Process of Design of such a Visual Notation

In the process of designing a visual language for ITL, there are several issues that gradually emerged. At first, a simple boxed notation for formulas and a circled notation for expressions was chosen. This was reported in [30]. The primary idea was to keep the notation simple and intuitive and experiment with an implementation of it. Another intention was obviously not to introduce any notation that can mislead the reader/user with his knowledge of any other visual language (formal/non-formal specification/programming language). The visual notation reported in the paper is reproduced below in Figure 4.3 and Figure 4.4 :

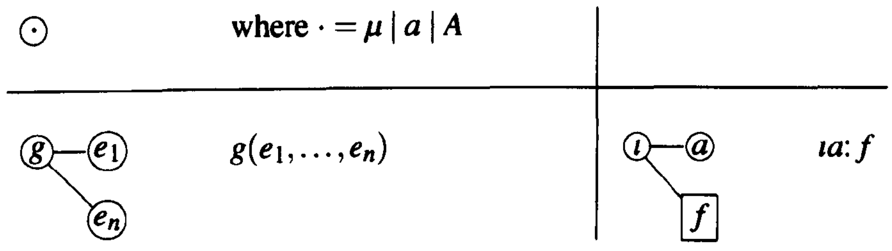


Figure 4.3: First Visual Notation for ITL Expressions

While an attempt was made to somehow have expressions in visual form, it did not turn out to be a feasible one. The expressions got complicated and introduced unnecessary processing of graphical information in an implementation without contributing to any gain in its understanding. Therefore, it was decided to introduce visual notations only for ITL formulae.

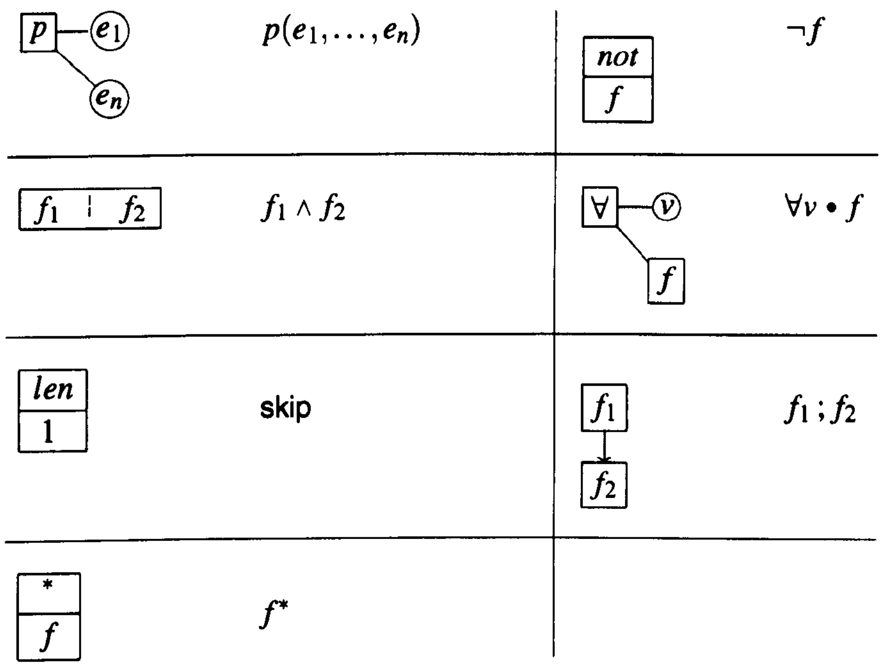


Figure 4.4: First Visual Notation for ITL Formulae

The primary considerations were therefore to keep the notation simple and intuitive and experiment with them.

Some of the visual notations for more frequently used ITL abbreviations are reproduced in Figure 4.5 from [30].

Some of the salient features of the above visual notation for ITL are :

1. The visual notation for the 'and composition' is the same as in Statecharts.
2. The visual notations for 'and' and 'chop' are simple and intuitive.
3. The visual notations for other constructs like sometime, always, next and so on, follow the same format as not, len, chopstar (*) in the figure thus aiding readability and adding simplicity to the formal specification.

## 4.6.3 Further Design Considerations

Any arbitrary visual notation could be chosen and given ITL semantics. An arbitrary notation can mean anything insensible also. Even among sensible notations, there can

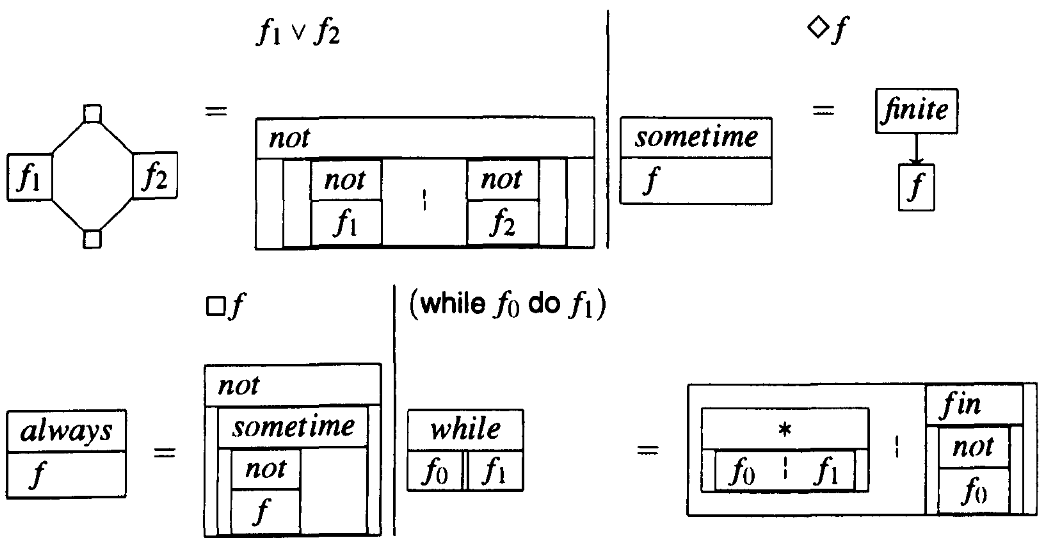


Figure 4.5: First Visual Notations for some Frequently used ITL Abbreviations

exist many possibilities. For example, [52] has several visualisations that various students suggested for representing an integer variable within the context of a program fragment. Figure 4.6 shows some of them. Many students, just like many program-

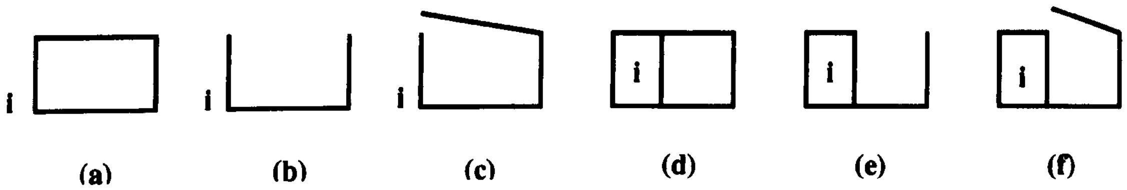


Figure 4.6: Various Ways of Representing “int i”

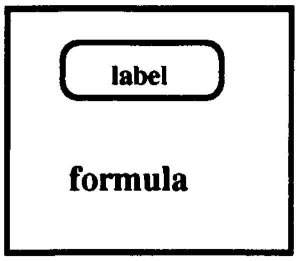
ming text books, came up with figures similar to (a). Those who chose (b) wanted to show that a value could go into and come out of an open box. Students who chose (c) programmed the lid to open to receive a value or pass a copy out and the lid was always shown slightly ajar to show that the value could change any time. Variable identifiers were mostly placed as shown in (a), (b) and (c) but also above, below, and to the right of the variable box. A minority of students chose to tie the identifier to the variable box as in (d), (e) and (f). For more details on the animations that were being tried in this context, the interested reader is referred to [52]. So, coming back to visual notations for ITL, as already mentioned, one could invent any notation. However, since the visual notation was being designed for ITL, a formal language, it was realized that there are fine points that could be addressed in order to make such a visual language more robust. In other words, in addition to being simple, intuitive and readable, we have to ensure that the semantics of the diagrammatic representations are uniquely defined with respect to any transformations on the diagram. We also have to specify how the diagram can be modified by operations like addition etc.. This is important to make sure that there are no confusions with respect to seemingly different visual representations of the same textual ITL formula. For example, a diagram on rotation should not only have the same semantics but also not lose its intuitive appeal.

Before these points are discussed further and addressed, a related problem with the notational format for temporal operators etc. above is now discussed.

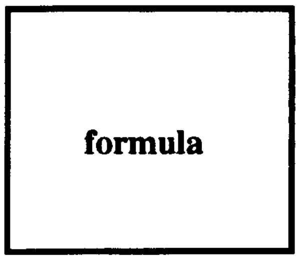
The notation for operators like always, sometime, length, chopstar etc. were all represented in a rectangular box with a horizontal line separating the formula in the box with a text at the top indicating the operator as shown in Figure 4.4 and Figure 4.5. While this notation is simple and readable, an objection to this was that it would be difficult to distinguish the textual formula in the box from the textual name of the operator. This was not a problem in implementation as the top portion had a text which had to be the operator name and the portion demarcated for the formula had the textual formula. Another criticism was how one would read such a formula if the box and its contents are inverted (except the text, of course), for example. Therefore, the following additional requirements for a visual notation for ITL had to be addressed:

- No confusions with regard to simple transformations of the diagram
- Semantics of operations like addition and deletion to a diagram had to be defined

A rounded box within the rectangular box was introduced to contain the operator information. When this is done as shown in Figure 4.7, the problem of giving a unique semantics to the notation is taken care of, if a transformation like rotation was performed, for example. The text within the rounded box inside the rectangular box is the operator while the text inside the rectangular box but outside the rounded box is the textual formula. For a simple formula with no operator like always etc., we can leave out the rounded box altogether to make the diagram look simpler. We call the text in the rounded box portion of the formula the "label".



(a)



(b)

Figure 4.7: The New Format for the Formula

Coming to the basic ITL constructs, just as we decided to do away with circled notation for expressions, we decided to write predicates, just as they are, in a rectangular box. At this point, however, it is necessary like to draw the attention of the reader to another interesting way of representing predicates. [130] describes work on visual notations which use visual cues to make the structuring of logical expressions more intuitive. The authors have used one of the more successful graphical metaphors in mathematics, the set inclusion, combined with other well known methods of representing relations graphically, like the graph formalism to reduce undue difficulties of use and interpretation of existing textual notations in the Horn clauses subset[2] of First Or-

[2]Horn formulas are practically important subclasses of formulas which have efficient ways of decid-

der Logic (FOL). In their visual logic, there are two different types of terms: FOL terms - defined as usual in FOL - that represent elements, and a new type of terms called set terms that represent sets of elements that satisfy a given predicate. The following is the textual definition of this new type of terms :

**Definition (Set Terms in the Visual Logic of [130])**

- A unary predicate $p$ is a set term.
- A predicate application $p(s_1,..,s_{n-1})$ is a set term, where $p$ is a n-ary predicate symbol $(p,q,r,..of\ arity\ n \geq 2)$ and $s_1,..,s_n$ are either FOL terms, set terms or set terms prefixed by an asterisk, '*s'.
- Given two set terms, $s_1$ and $s_2$, $s_1 \cap s_2$ and $s_1 \cup s_2$ are also set terms.
- Given a set term $s$, $\bar{s}$ is a set term.
- Nothing else is a set term.

Figure 4.8 shows examples of terms in their visual logic. A predicate is represented as a square box with the predicate symbol on a corner, while a function is represented by a rounded box, also with the function symbol on a corner. Variables are represented as circles whereas constants are represented in a rounded box with the constant symbol, for example $a$, in Figure 4.8, on a corner. The difference between a function and a constant is that there are no arrows leading to the rounded box for a constant. Only arrows can originate from a rounded box for a constant. A double arrow is used where the textual form contains an asterisk, '*s'. It can be seen that the visual representation for likes (*man) i.e., "likes every man", contains a double arrow, whereas intersecting boxes are used to depict intersection as in "likes some tall men". For additional details, the reader is referred to [130].

ing their satisfiability. 'Horn' is derived from the logician A.Horn's last name

While it is a very interesting way to represent predicates, we still stick to representing them textually within a rectangular box in order to avoid the proliferation of many graphical primitives.

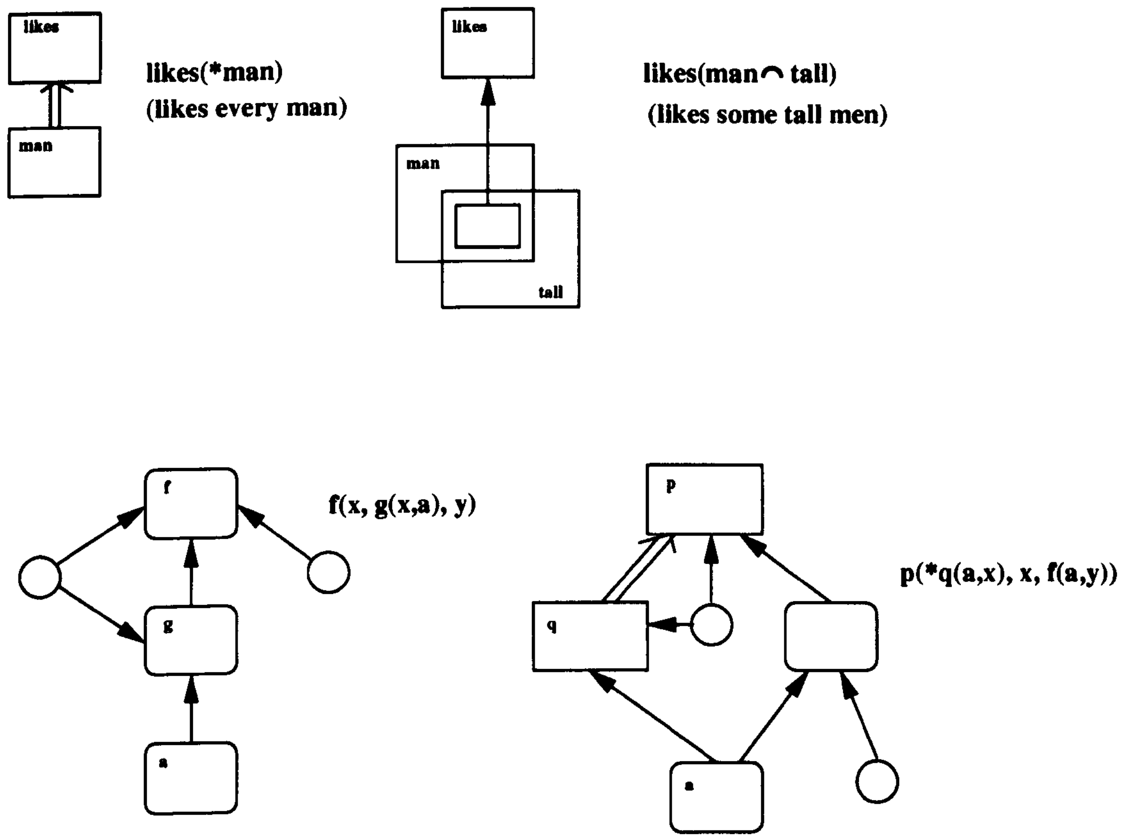


Figure 4.8: Examples of Terms in the Visual Logic of [130]

The discussion of other changes in visual notations for ITL is given below.

The negation has to involve, either the label "not", or some substitute for it. One possibility is to cross out the whole box indicating negation. Another is to use a cross in the label portion of the rectangular box. The latter is preferable as it follows the format being used and keeps the formula clean and readable as shown in Figure 4.9. An intuitive way to represent "chop" would be to have two boxes, one corresponding to the resulting left sub-interval, and another corresponding to the right sub-interval with an arrow from the former box to the latter as shown if Figure 4.10. The direction of the arrow indicates that the formula enclosed by the box on which the arrow originates corresponds to the formula that is true on the left sub-interval. Since chopstar should

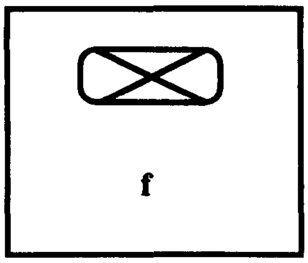


Figure 4.9: Negation of a Formula

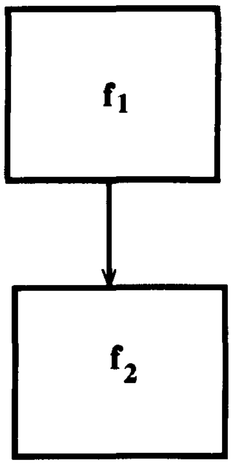


Figure 4.10: The Chop

suggest that a formula repeats i.e., loops in a way according to its definition, Figure 4.11 is a good, intuitive choice for it.

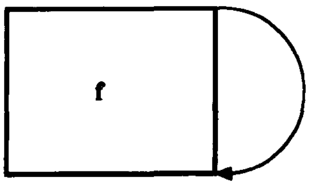


Figure 4.11: The Chopstar

### 4.6.4 The Syntax

The previous sections discussed some of the ideas that influenced the choice of our visual notations. In the definition of a visual language, the choice of an appropriate visual syntax is very important. Figure 4.12 summarises the syntax of our visual notation for primitive ITL constructs. In the previous section, the rationale for the visual notation has already been covered. The salient features of the notation can be summarised below as follows:

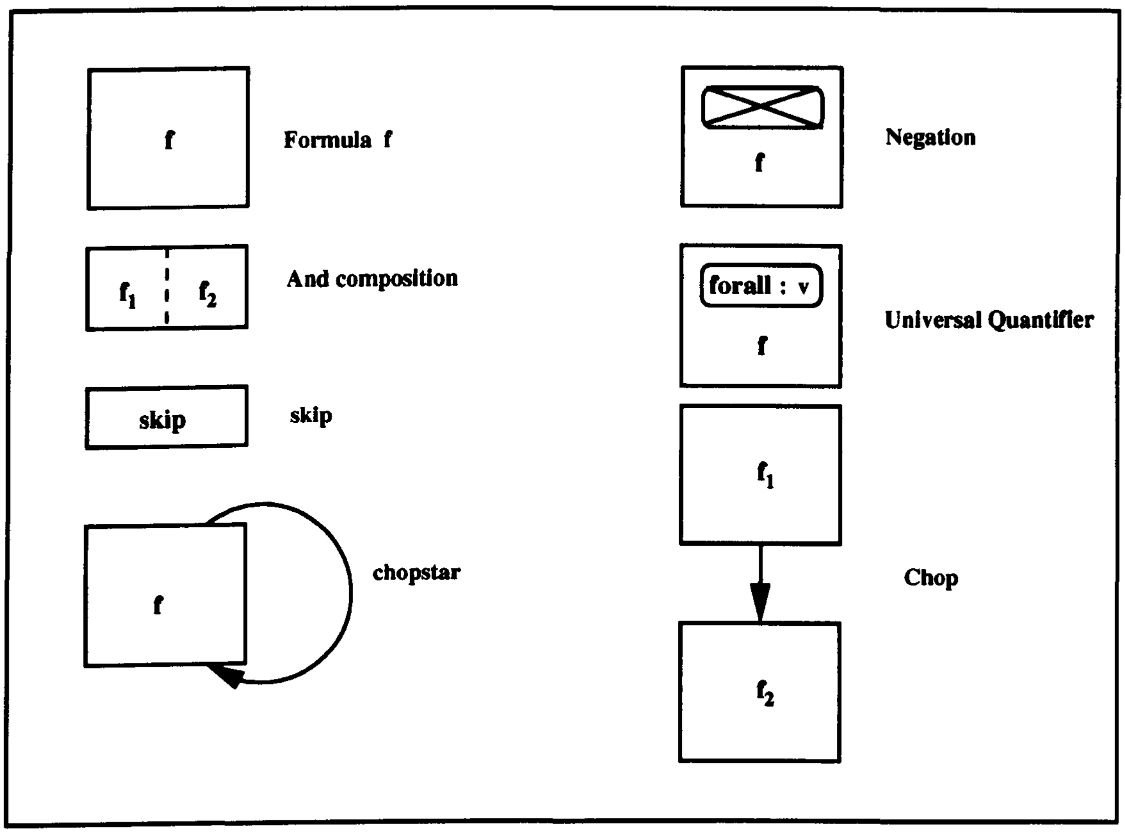


Figure 4.12: Final version of the Visual Notation for Primitive ITL Formulae

- Expressions are written as in textual ITL.
- A formula is enclosed in a rectangular box which also contains a rounded rectangular box for holding any operator information ; the rounded rectangular box maybe omitted if there is no operator involved.
- The "And composition" has dotted lines between the components in the formula.
- The negation has a cross mark in the label portion of the rectangular box.
- The existential quantifier also follows the labelled rectangular box format.
- The chop has a directed arrow between the two rectangular boxes containing the formulae for the left and right subintervals respectively.

- The chopstar has a directed arrow from the rectangular box to itself indicating a looping construct.

### 4.6.5 Visual Notations for Frequently used Abbreviations

So far, visual notations have been introduced for the primitive constructs. Now the other constructs of ITL can be expressed visually using the basic primitive visual constructs in Figure 4.12. This is done for the convenience of the user. In other words, we have a meta visual language layer above the primitive visual language layer primarily for user convenience and visual appeal. As an additional possibility, we will allow the user to define his/her own convenient visual representations for any constructs (or parts of specifications) and thereby customise the visual notation. Such customisation will be allowed, however, with specified guidelines so that the language conforms to our design principles.

Using the visual syntax for a formula with a label, we can define the temporal constructs "sometime" and "always" as in Figure 4.13 and Figure 4.14 respectively. The

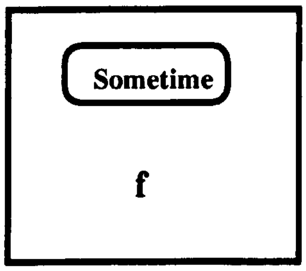


Figure 4.13: Sometime

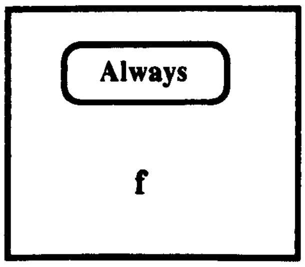


Figure 4.14: Always

"Or" is defined as in Figure 4.15 because the branches coming out from a filled circle in the box indicate, intuitively, that one of them is to be selected (as "true"). Figure

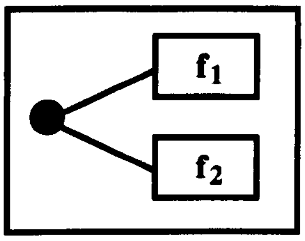


Figure 4.15: Or

4.16 shows the construct for implication. One of the arguments is shown in a shaded box "sort-of suggesting" something more detailed. By enclosing the left argument of $f_1 \supset f_2$ in a shaded box, we mean that $f_1$ is a more detailed formula.

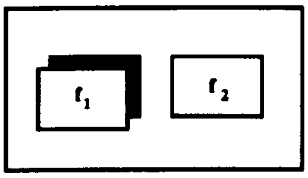


Figure 4.16: Implication

Figure 4.17 summarises some frequently used visual ITL abbreviations.

### 4.6.6 Visual Notations for Concrete Constructs

The "While" concrete construct could be defined as in Figure 4.18 as the label suggests a "while" and the user visualises two arguments to it, one to keep "repeating" while the other is true. The box in the left portion of the "double line separation" contains the first argument to check while the box in the right portion is the second argument. An explicit arrow could also be added in the double line portion to suggest the direction of reading as in Figure 4.19. An explicit arrow was chosen as this removes any chance for misinterpretation even when the figure is turned upside down, for example. Instead of the straight arrow we used, we could have other possibilities as shown in Figure 4.20 to suggest that it is a looping construct. However, I prefer to have as few graphical

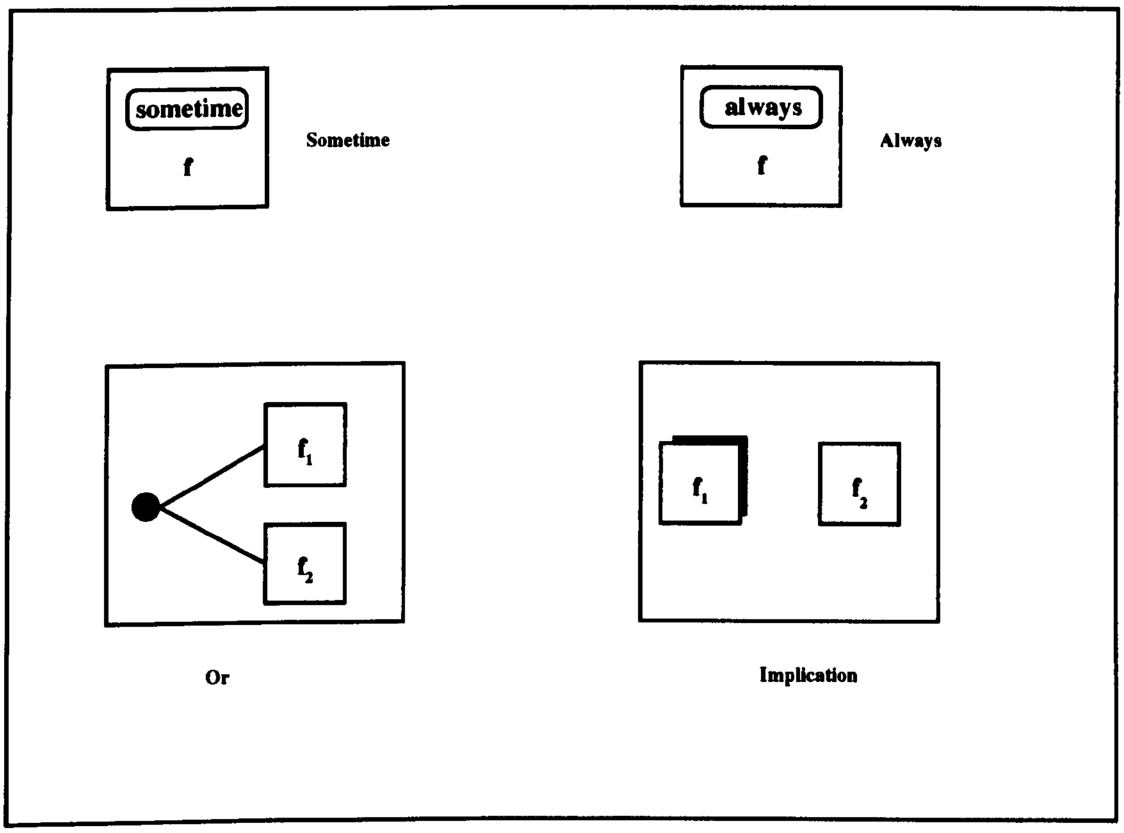


Figure 4.17: Summary of the Visual Notation for Some Abbreviations

primitives as possible for the visual construct in order not to crowd it for the reader as well as for the tool that implements it. So, the notation in Figure 4.19 is chosen. Also, we annotate the arrow optionally as shown with $[t_m]$ to depict the execution time for the body of the while loop. Omitting this annotation means an execution time of 1. However, we could explicitly add [1] for unit execution time. Instead of the straight arrow we used, we could have used other possibilities as shown in Figure 4.20.

Figure 4.21 shows the visual constructs for the concrete "If..then" constructs.

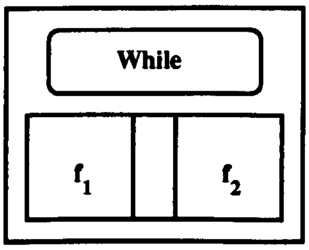


Figure 4.18: While

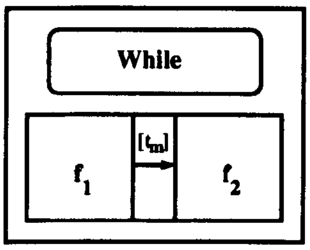
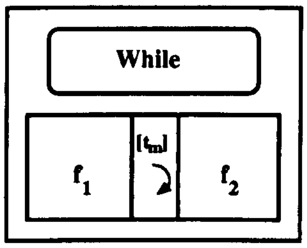


Figure 4.19: While2



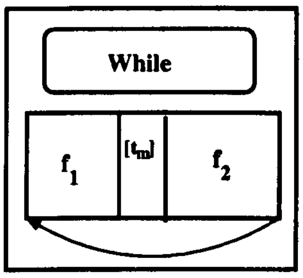


Figure 4.20: While3

[90] is a good reference to demonstrate the flow of control notations with respect to pancode and boxchart notations which are visually-oriented programming language templates. The dynamic mental images evoked by the Boxchart notation and the Pancode notation make program behaviour easy to understand.

Also, we optionally include a diamond box inside the visual formula to indicate that the formula is expandable for details. A "+" inside the diamond indicates that zoom-in is possible whereas a "-" indicates that zoom-out is the only possibility. The default case is where there is nothing to zoom into, in which case the diamond box is omitted. This is illustrated in Figure 4.22.

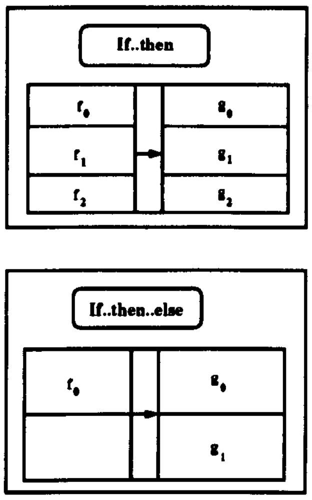


Figure 4.21: If Then Concrete Statements

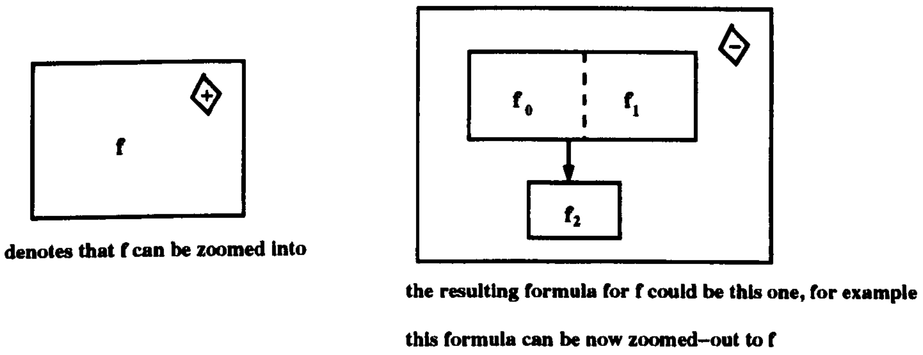

f
denotes that f can be zoomed into
$f_0$
$f_1$
$f_2$
the resulting formula for f could be this one, for example
this formula can be now zoomed–out to f

Figure 4.22: Denoting Zoom

### 4.6.7 Additional Concrete Constructs for Timing constraints, Resource Allocation and Concurrency

In [27], shortcomings of ITL as a formalism in dealing with explicit expressibility of timing constraints, resources and concurrency were identified. These were addressed in [29] by providing a timed-communication model allowing explicit representation of concurrency, resources and timing. This model is very closely related to that of the Temporal Agent Model (TAM) [135] which is a well-established formalism for the development of real-time safety-critical systems [105]. For a description of the timed communication model, the reader is directed to appendix A.

There follows an example of the TAM concrete constructs.

- **Channel communication**

  Synchronous communication links are called channels where read and write occur at the same time.

  *Channel C in P* introduces a new channel within *P*.

  $C!e$ denotes an output agent that sends the value of expression $e$ over $C$.

  $c?x$ denotes an input agent that stores the value received over $C$ in $x$.

  $C.x$ denotes the value of $x$ in $C$.

  Figure 4.23 presents the TAM constructs using the visual notation for channel communication with informal semantics. The textual formulae involves "!", "?"

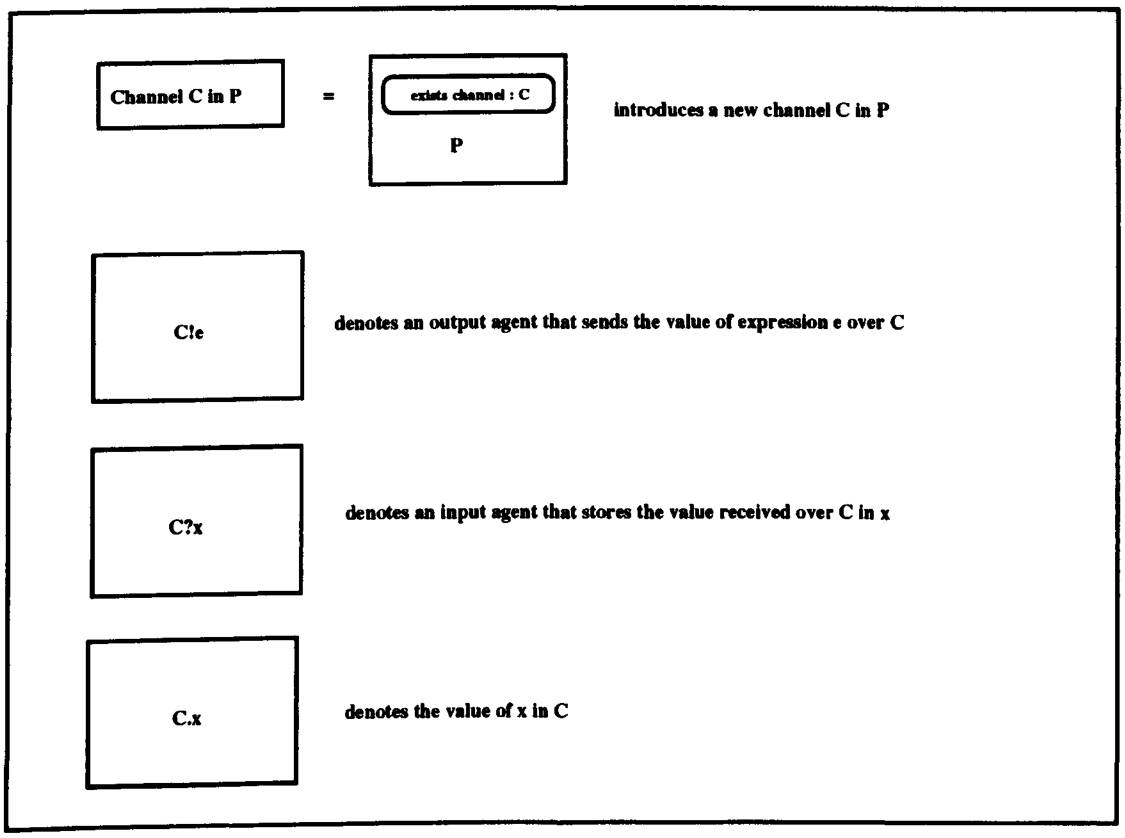

Channel C in P
=
exists channel : C
P
introduces a new channel C in P
C!e
denotes an output agent that sends the value of expression e over C
C?x
denotes an input agent that stores the value received over C in x
C.x
denotes the value of x in C

Figure 4.23: Channel Communication Constructs

and "." and the current visual notation merely encloses formulae containing those in boxes. I think, we could do more, especially during implementation, by utilising the possibility of using the optional label within the formula box. Within the label, we could use a suitable icon to introduce visual cues regarding these symbols. The fact that we can do more to a visual language during implementation is one additional advantage over a purely textual notation. Figure 4.24 shows one such possibility. The optional use of icons within labels could be made as a

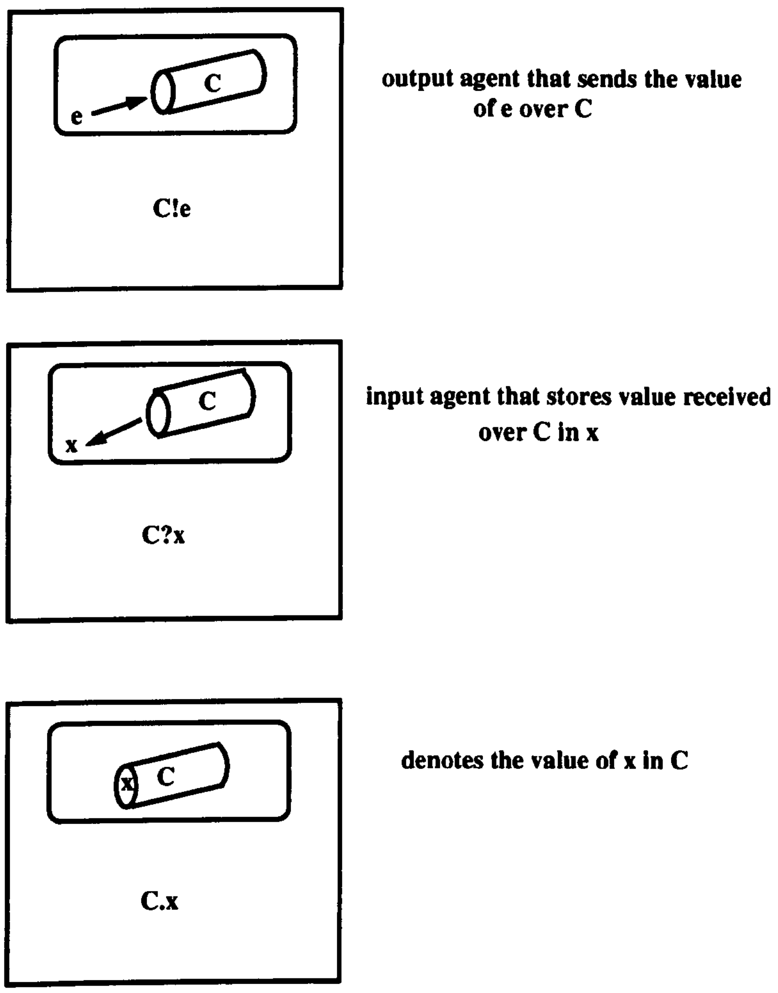

e
C
C!e
output agent that sends the value of e over C
x
C
C?x
input agent that stores value received over C in x
x
C
C.x
denotes the value of x in C

Figure 4.24: Channel Communication Constructs with Icons in Labels

feature of the tool implementing VisITL.

- **Shunt communication**

  Shunts are defined as asynchronous communication links in appendix A.

  Variables $s$ represent shunts whose values are tuples $(t, v)$ where $t$ is a stamp and $v$ is the value written. The stamp value of $s$ is denoted by $\sqrt{s}$ and the value stored in $s$ by $read(s)$.

The agent *Shunt s in P* introduces a new shunt $s$ within $P$.

The agent $(x,y) \leftarrow s$ performs an input from shunt $s$ storing the value in $y$ and the timestamp in $x$.

The agent $v \rightarrow s$ writes the current value of expression $v$ to shunt $s$, increasing the stamp by one.

Figure 4.25 presents the TAM constructs for shunt communication with informal semantics.

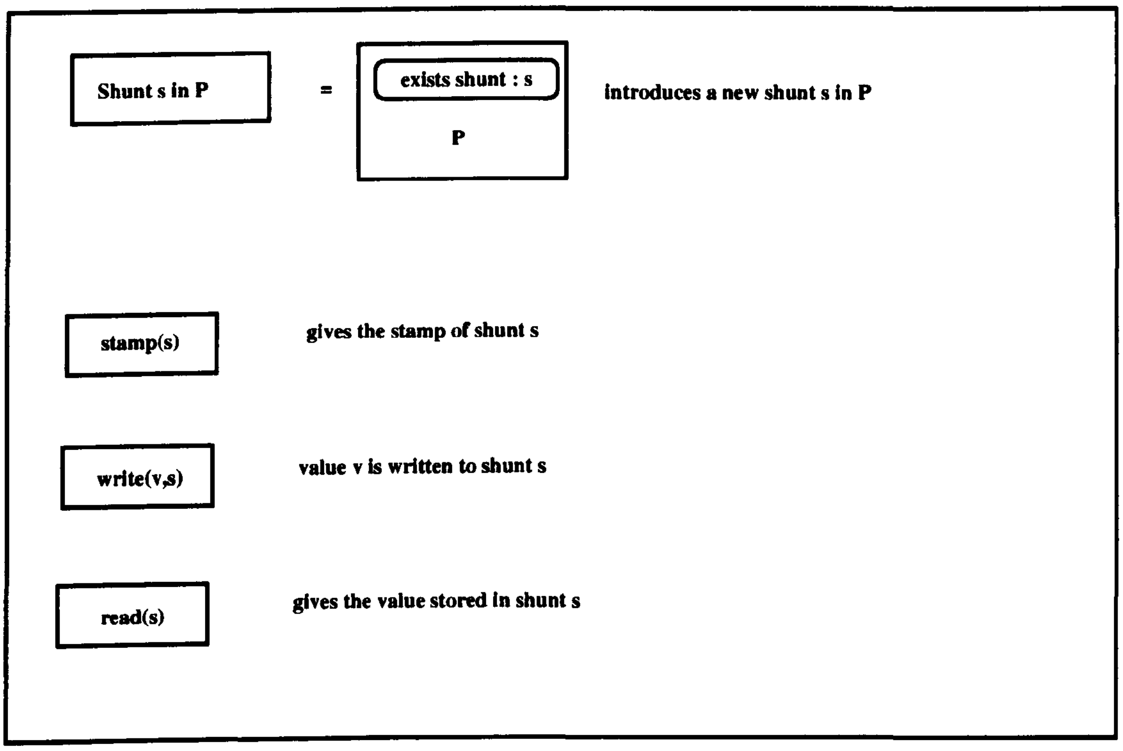

Shunt s in P
=
exists shunt : s
P
introduces a new shunt s in P
stamp(s)
gives the stamp of shunt s
write(v,s)
value v is written to shunt s
read(s)
gives the value stored in shunt s

Figure 4.25: Shunt Communication Constructs

- **Delay and timeout**

$delay_n$ describes an agent which first holds up for $n$ time units and then terminates with all global variables untouched.

The notation $P \trianglelefteq_d Q$ denotes an agent which behaves like $P$ if $P$ is executed within $d$ time units, and like $Q$ otherwise. Figure 4.26 presents the TAM constructs for delay and timeout.

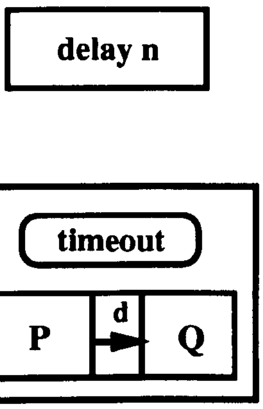


Figure 4.26: Delay and Timeout Constructs

- **Resource Allocation**

  *request*$(v, res)$ denotes an agent that requests $v$ units of resource $res$. If these $v$ units are unavailable, the agent waits for them. *release*$(v, res)$ denotes the agent that releases $v$ units of resource $res$. Figure 4.27 presents the TAM constructs for resource allocation with informal semantics.

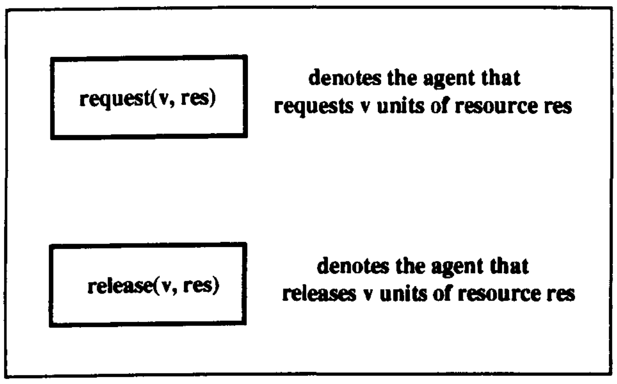

request(v, res)
denotes the agent that requests v units of resource res
release(v, res)
denotes the agent that releases v units of resource res

Figure 4.27: Resource Allocation Constructs

### 4.6.8 Geometrical Guidelines on Constructing VisITL Specifications

As far as operations like addition, deletion etc. are concerned, we give rules/guidelines as to what constitutes a meaningful visual specification. Some of them are enumerated below :

1. No box can be added such that it overlaps another.

2. Deletion of the rounded box within the rectangular box is allowed, and recommended, if the label portion is empty.

3. For the chop construct, there are two boxes on each end of the directed arrow ; none of them can be independently deleted leaving a dangling directed arrow.

4. There cannot exist any bits and pieces of formulae that are not connected to each other through the "And" composition or the "chop" primitives unless at the meta level layer, they are connected through some abbreviations like the "Or" operator. Hence any additions or deletions performed should conform to this constraint in the final complete diagram.

## 4.7 Summary on the Choice of Visual Notation

The choice of the visual notation has been influenced by various factors as seen in section 4.6. These influences were a direct result of the the important design features identified for visual representations from section 3.2. The following paragraph summarises some key points.

In section 3.1.11, we saw how an Interval Logic, namely GIL [41], depicted formulae in the logic visually. It is dependent on constructing different sub-intervals and then depicting the predicates that are true in such subintervals. Hence, it needed different kinds of graphical primitives like a solid line to represent strong intervals i.e., non-empty intervals, a double solid line to depict a weak interval, a single arrowhead to indicate a weak search while an interval is constructed, a double arrowhead to indicate a strong search and so on apart from using temporal operator symbols in the visual representation. Also, the placement of the predicate relative to the line depicting the interval has a meaning ; for example, if it is the middle of the interval, then, it is an invariant as in Figure 3.7(b). In the context of VisITL language, a much simpler, more intuitive and readable language was needed which also integrated TAM for concrete communication constructs.

It was also considered in section 4.6.3 how predicates were depicted using set theory concepts in a visual logic in [130]. This logic, being based on set theory, has graphical primitives like overlapping boxes which are unnecessary in VisITL, and circles for variables and constants additionally which we found unnecessary as it over-crowds the visual representation as well as leads to unnecessarily complicating the implementation. Moreover, this visual logic is not a temporal logic.

We also saw in chapter 3 how parallel states were depicted in statecharts-based formalisms. The same "dotted-line"representation was chosen for the "and-composition" in VisITL so that no new notation is introduced for similar concepts in other languages. This would help avoid any confusion for users previously accustomed to other languages.

For the ITL "chop", a line with arrow was chosen to represent the meaning of chop i.e., sequential composition. For "chopstar", a loop was depicted around the box. The label in the box was used for readability. The concrete constructs like "While", "If Then" etc. also followed a similar format. This is true for the TAM communication constructs as well. Hence, VisITl was not only made simple, intuitive, readable and unambiguous but also one that integrated abstract constructs and concrete constructs in similar notations. This aids communicativeness between users at all abstraction levels.

The geometrical guidelines for constructing VisITL specifications show how a specification can be manipulated and composed. The VisITL abbreviations allow us to manipulate diagrams to suitable equivalent forms. In a similar way, the user can add new abbreviations as a way of extending the syntax and thus customise the notation. Chapter 6 demonstrates the scalability of this approach. Chapter 7 will demonstrate the realisability of this approach. The VisITL language derives its expressivity from ITL.

The salient features of the visual notation can be summarised as follows :

The visual notations for Negation, Chop and Chopstar use simple intuitive concepts to depict the meaning of the formula. The "And-composition" uses dotted lines similar to statecharts-based formalisms. The abstract and concrete constructs both follow the box-format. A label within the box aids readability. To be more visual, the label could also hold icons as in the case of TAM constructs. All these constructs, as we saw, adhere to our design principles identified in section 3.2.

## 4.8 Collection of Visual Constructs Introduced

This section simply presents the already introduced visual constructs in one place for the convenience of the reader.

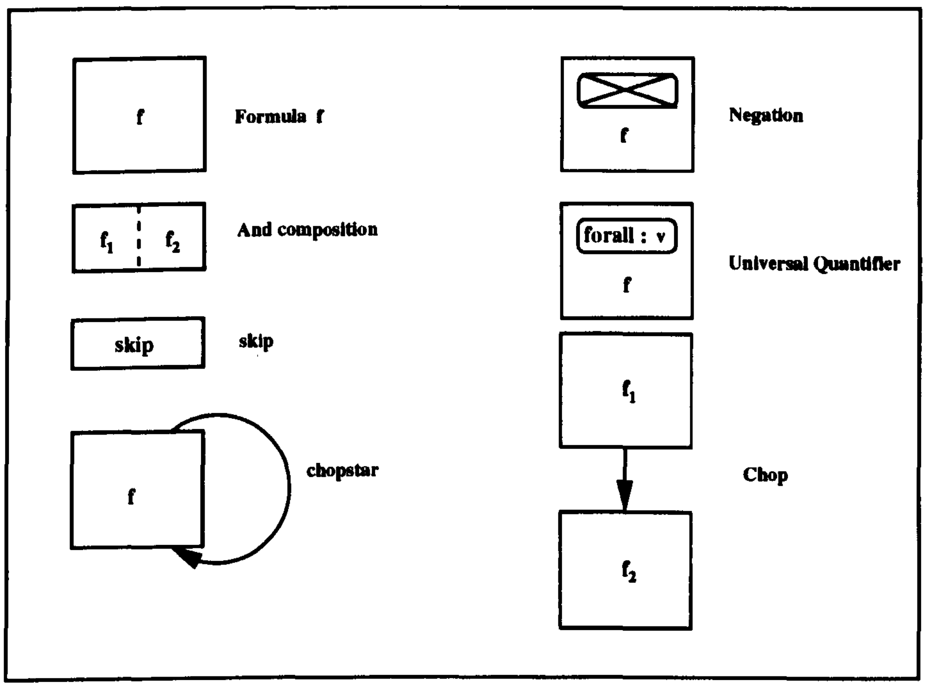


Figure 4.28: Primitive Visual ITL Formulae

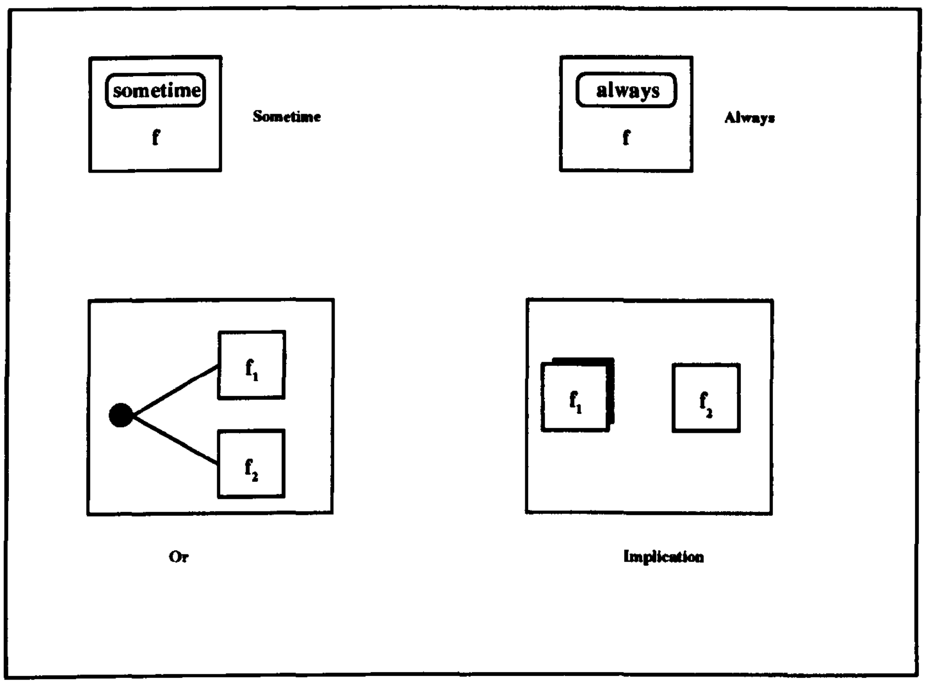


Figure 4.29: Frequently used Visual Abbreviations

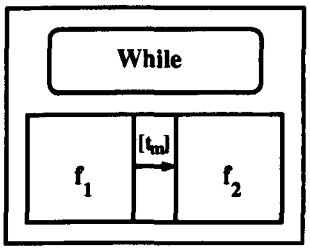


Figure 4.30: The Concrete Visual Construct for While

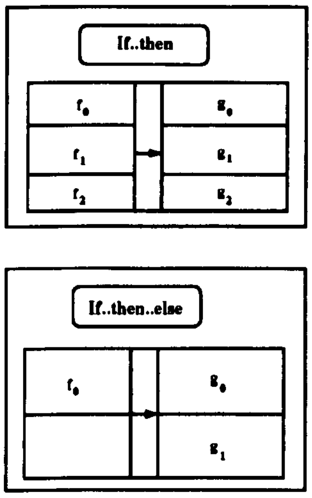


Figure 4.31: The Concrere Visual Constructs for If Then, If Then Else

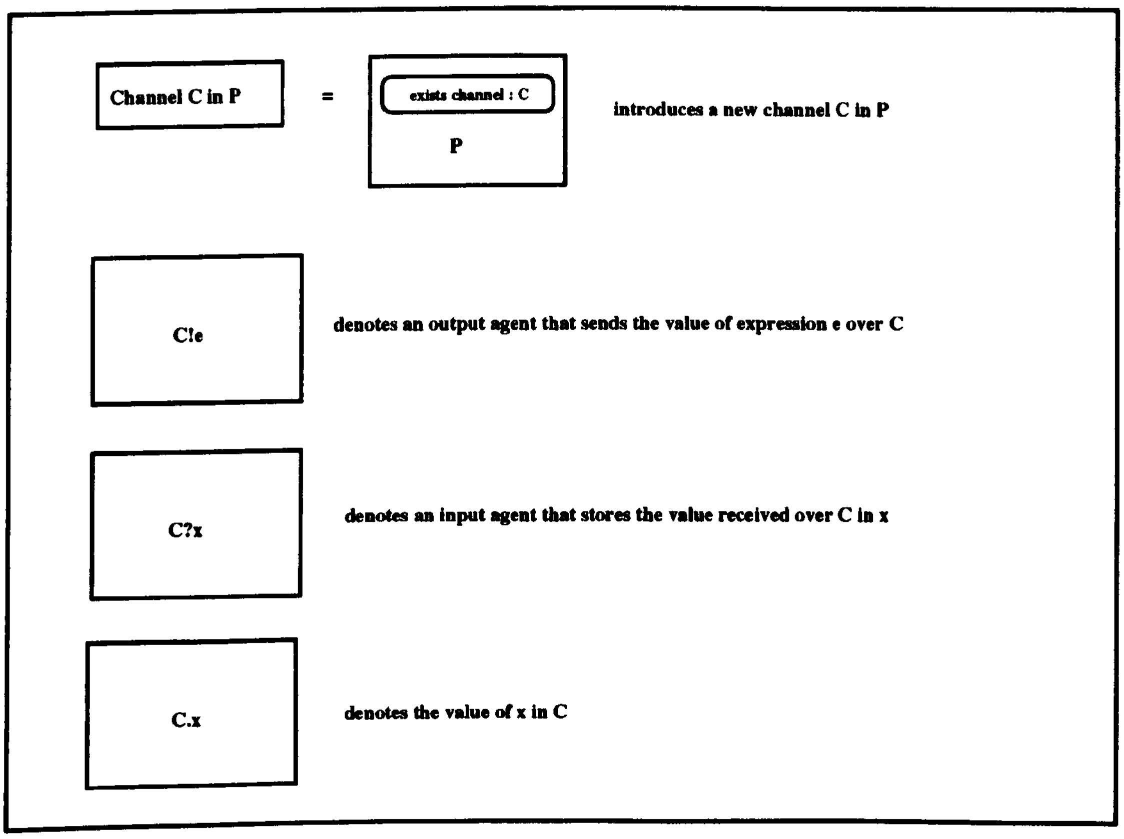

Channel C in P
=
exists channel : C
P
introduces a new channel C in P
C!e
denotes an output agent that sends the value of expression e over C
C?x
denotes an input agent that stores the value received over C in x
C.x
denotes the value of x in C

Figure 4.32: The Visual Channel Communication Constructs

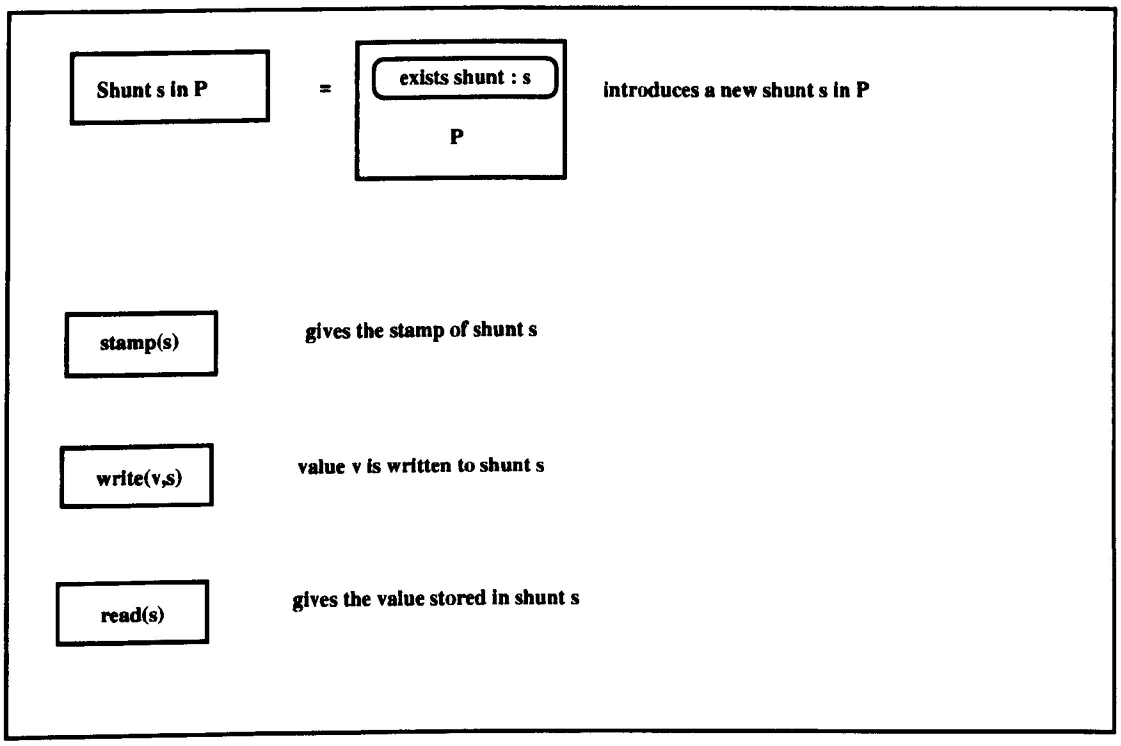

Shunt s in P
=
exists shunt : s
P
introduces a new shunt s in P
stamp(s)
gives the stamp of shunt s
write(v,s)
value v is written to shunt s
read(s)
gives the value stored in shunt s

Figure 4.33: The Visual Shunt Communication Constructs

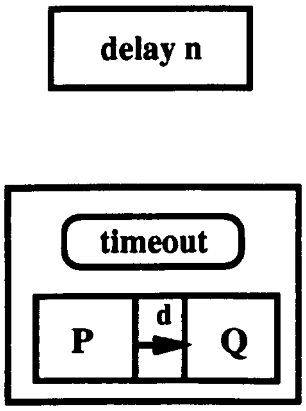


Figure 4.34: The Concrete delay and timeout Constructs

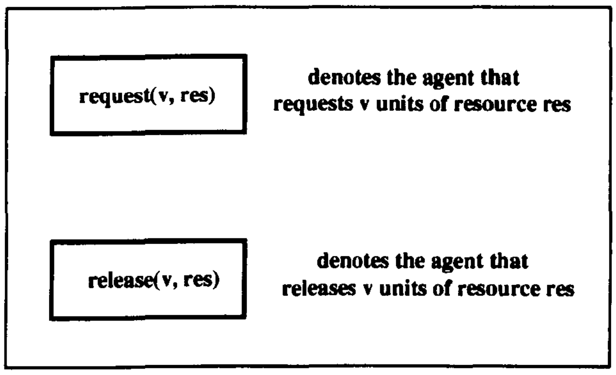

request(v, res)
denotes the agent that requests v units of resource res
release(v, res)
denotes the agent that releases v units of resource res

Figure 4.35: The Concrete Resource Allocation Constructs

## 4.9 Examples in Visual Notation

In this section, simple examples are given. The Automobile Cruise Control System introduced in section 4.2.6 is also re-worked using the VisITL notation.

### 4.9.1 Some Abstract Examples

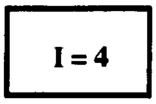


Figure 4.36: Simple Examples

Figure 4.36 is equivalent to $(I = 4)$. Figure 4.37 (a) is equivalent to $\bigcirc(I = 4)$. This means that $I$ takes the value of 4 in the next state. Figure 4.37 (b) means that $I$ takes the value of 4 in all the states of the interval. Its ITL equivalent is $\Box(I = 4)$. Figure 4.37 (c) means that $I$ takes the value of 4 in some one/more states in the interval. Its ITL equivalent is $\Diamond(I = 4)$. Figure 4.37 (d) means that $I$ takes the value of 4 in the next state if there is one. Its ITL equivalent is $\textcircled{w}(I = 4)$. Figure 4.38 (a) is equivalent to the ITL formula $(I = 1) \wedge \bigcirc(I = 0)$. Figure 4.38 (b) is equivalent to the ITL formula $(I = 1) \wedge \Diamond(I = 0)$. Figure 4.39 (a) is satisfied by an interval if (I=0) is true in the left subinterval and (I=1) is true in the right subinterval. Its ITL equivalent is $I = 0\,;I = 1$.

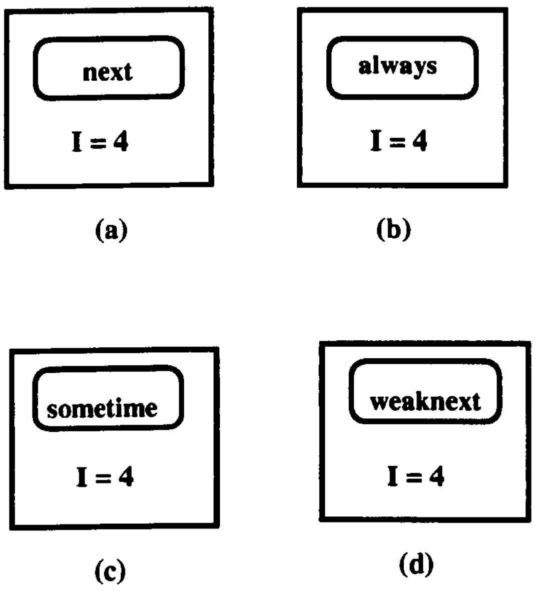


Figure 4.37: Simple Examples with Different Formula Labels

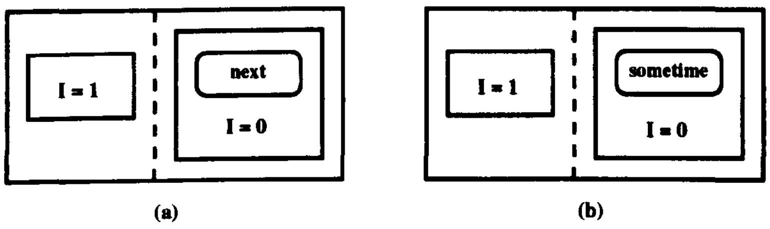


Figure 4.38: Simple Examples using Parallel Composition

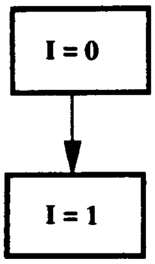


Figure 4.39: Simple Example using Chop

### 4.9.2 An Example of An Automobile Cruise Control System

Let us consider the **automobile cruise control system** again.

We can now specify a constraint on the state space as below :

**If the value of the speed is either Zero or Max., then, SpeedState is going to be constant**

This is specified in Figure 4.40.

Implicit in this constraint is the fact that the **SpeedState** may or may not be constant when the **SpeedVal** is neither Zero or Max.

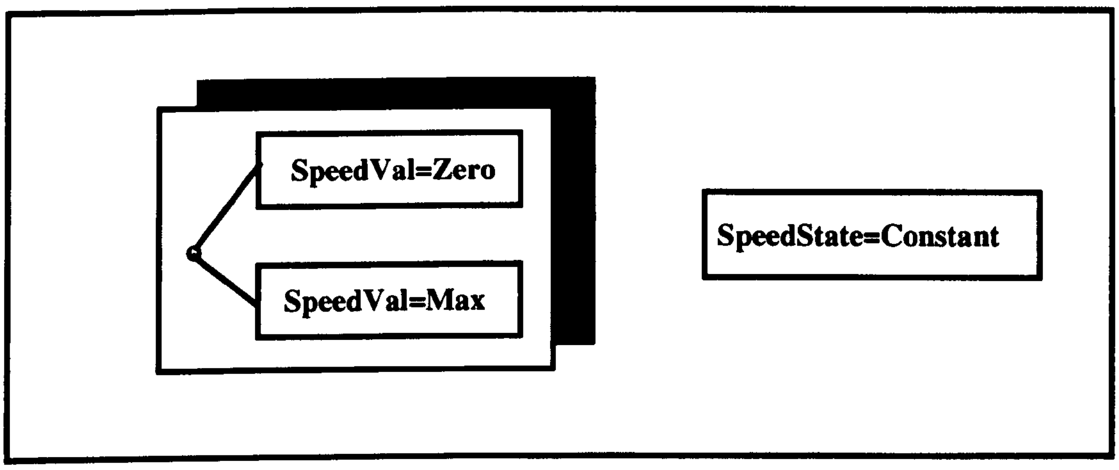


Figure 4.40: Constraint3

**The Initial State Specification**

We can define a formula, *FInitial,* for initial states of the cruise control system. The initial states include, say, the engine being off, the cruise lever set to off, the car speed being zero, the internal state being idle and the desired speed not yet being set. This can be specified in VisITL as in Figure 4.41.

**Safety and Liveness Specification**

**If the engine is off, then, the car will eventually stop**

Turning off the engine implies that eventually, the speed will become zero.

The spec. is given in Figure 4.42.

**If the lever of cruise const is applied, and the brake is off at that time, the**

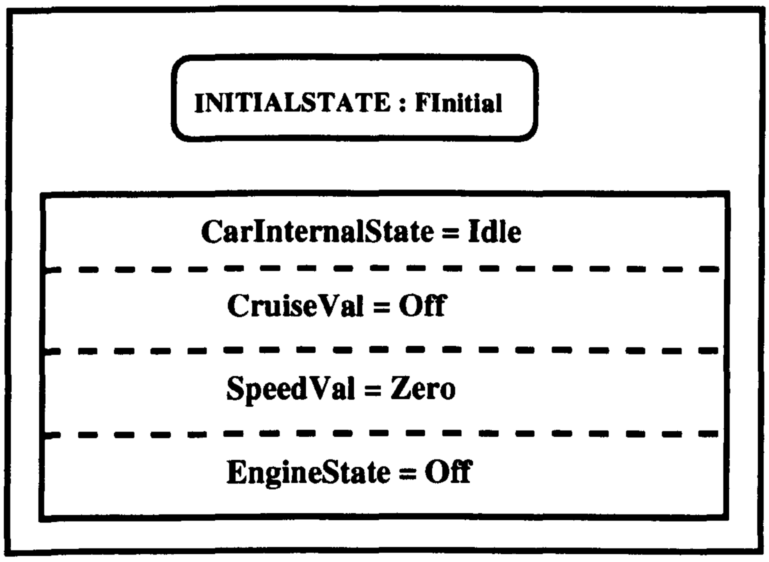


Figure 4.41: Initial State

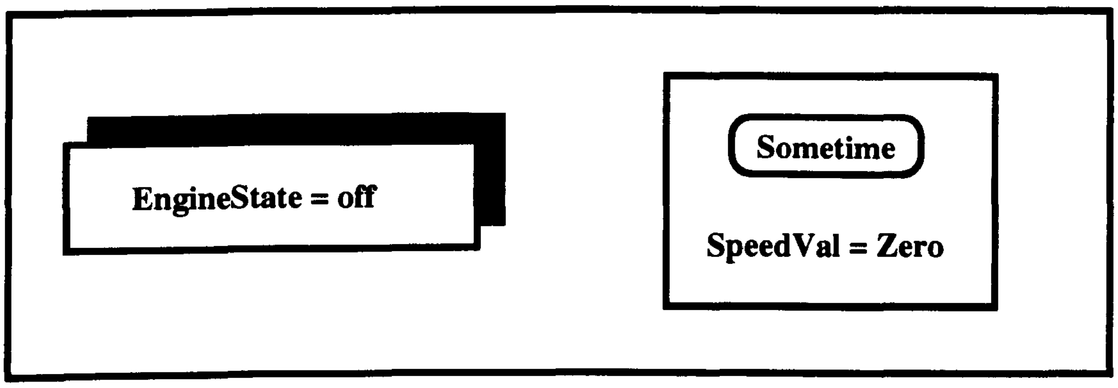


Figure 4.42: Liveness1

**desired speed, DesiredSpeedSetting, will eventually be set, whether it has been set previously or not.**

The spec. is given in Figure 4.43.

**Examples with real time constraints :**

Let us first make some assumptions and introduce certain constants. Let *CAcceleration* denote the magnitude of a constant value of both acceleration and deceleration rate. The maximum value for the car speed was already introduced as *Max*. Hence the maximum time required for a car to achieve its maximum speed, say CMaxTime, will be Max/CAcceleration. Let CInfinity denote an arbitrary large number for time greater

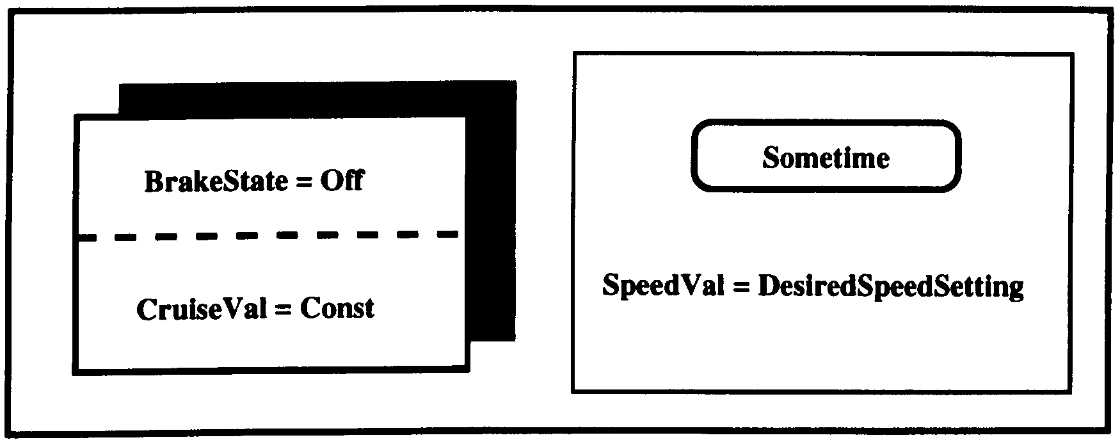


Figure 4.43: Liveness2

than CMaxTime.

Now we can introduce an example specification as in Figure 4.44 with real-time constraints.

**If the current speed is below the desired speed, and if the car increases its speed continuously for a period longer than CMaxTime, then the speed of the car will eventually reach the desired speed.**

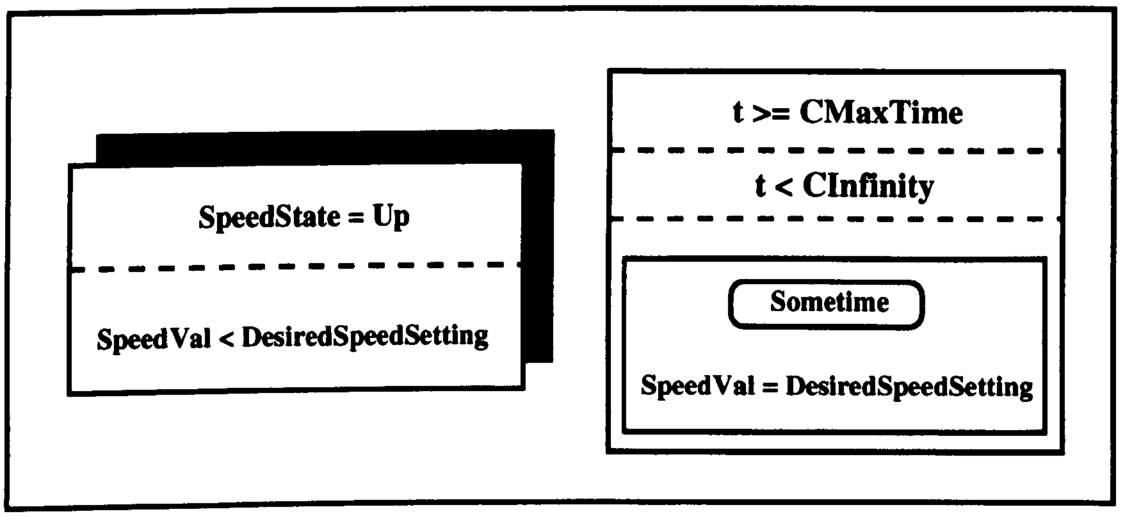

SpeedState = Up
SpeedVal < DesiredSpeedSetting
t >= CMaxTime
t < CInfinity
Sometime
SpeedVal = DesiredSpeedSetting

Figure 4.44: Real1

## 4.10 Chapter Summary

In this chapter, we have introduced ITL with example specifications highlighting some of the problems with textual ITL. A visual notation for ITL has been introduced after elaborating on the design rationale. Several example specifications have been shown visually. However, just having a formal and visual language is not enough for formal development. There is a development method needed to derive an implementation from the VisITL specification. A development method should also support the derivation of a high-level specification from an existing implementation. Therefore, we need a **Visual Framework** for formal systems development that not only consists of a formal and visual language but also rules for transforming VisITL specifications suitably to either make it more concrete or more abstract depending on whether the user wishes to derive an implementation or a high-level abstract specification. In the next chapter, we shall explore how the development process in VisITL is made possible.

# Chapter 5

# A Visual Framework for VisITL

The main objective of this chapter is to introduce the Visual Framework for VisITL.

## 5.1 The Visual Framework

In chapter 4, the VisITL language was introduced with examples. Although this visual language captures the requirements formally, there is a need for a development process in which an implementation can be derived formally. Moreover, in order to redevelop or understand already existing implementations, there is a need to extract a high-level specification from the implementation. In other words, the *Development Process* aims to incorporate the possibility of *Redevelopment* of existing implementations. Therefore, we need a suitable formal framework. This is defined as the **Visual Framework for Formal Systems Development using Interval Temporal Logic** or simply the **Visual Framework for VisITL**. As explained below, it consists of the following components:

- The VisITL visual and formal language
- Visual Refinement Rules for deriving an implementation
- Visual Abstraction Rules for deriving a high-level specification from an existing implementation

The Figure 5.1 depicts the development process using VisITL. In this **Visual Framework**, apart from the VisITL language, there are visual refinement rules and abstrac-

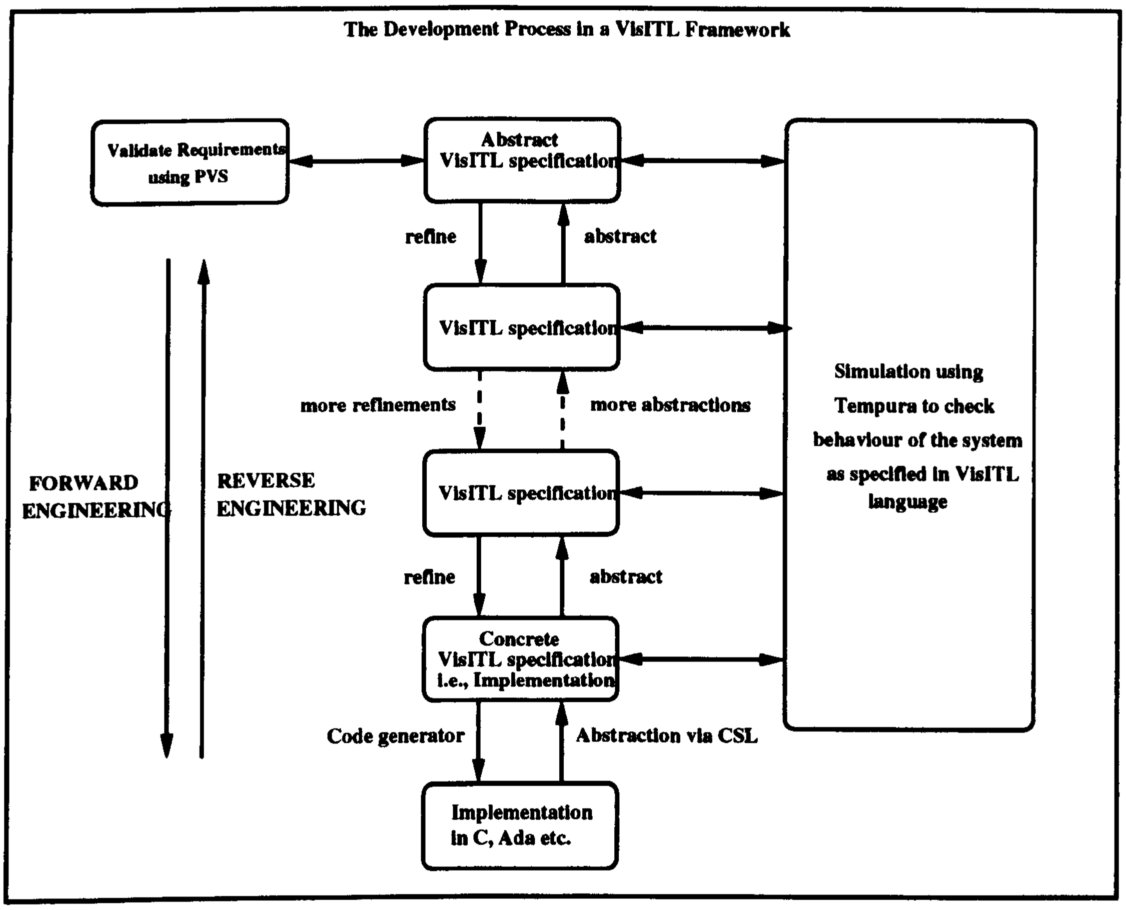

The Development Process in a VisITL Framework
Validate Requirements using PVS
Abstract VisITL specification
refine
abstract
VisITL specification
more refinements
more abstractions
FORWARD ENGINEERING
REVERSE ENGINEERING
VisITL specification
refine
abstract
Concrete VisITL specification i.e., Implementation
Code generator
Abstraction via CSL
Implementation in C, Ada etc.
Simulation using Tempura to check behaviour of the system as specified in VisITL language

Figure 5.1: The Development Process in a Visual Framework using VisITL

tion rules for systems development using VisITL specifications. Initially, with a VisITL specification capturing requirements, the user could perform validation using the proof system of ITL. Also, the user would be able to simulate the system behaviour at any abstraction level using Tempura. To allow the user to seamlessly go forward from the abstract VisITL specification, we need to provide sound visual refinement rules in the framework. In other words, just having a VisITL specification capturing requirements is not enough. To derive an implementation i.e., forward engineer the specification, we need visual refinement rules in the framework. Similarly, in order for the user to perform reverse engineering [103], i.e. derive a high-level specification from an existing implementation, we need to provide sound abstraction rules in the VisITL framework.

This chapter is therefore devoted to the development of visual refinement and abstraction rules. As the Figure 5.1 depicts, a concrete VisITL specification could be translated into an implementation in a suitable language like Ada, C etc.. In reverse engineering, the source code in C or Ada etc. is first translated into a Common Structural Language (CSL) [103] and then into a Concrete VisITL specification.

## 5.2 Refinement

This section will introduce refinement and visual refinement rules.

### 5.2.1 An Overview of Refinement

The term *refinement* is used in several related but subtly different ways in technical contexts [87]. One usage which is relevant to us is it being used as a relation on specifications and treating an implementation as a special case of a specification. In simple informal terms, we could view the process of refinement as adding some detail or improvement to a specification to obtain a new specification. This way, implementations can be derived from specifications incrementally in a stepwise manner.

For untimed systems, a refinement calculus was developed by Back [7]. It extends Dijkstra's weakest precondition semantics for total correctness of programs between program statements. The refinement calculus was extended to provide a framework for total correctness for parallel systems in [8, 9]. Morgan's work [112] details how refinement calculus could be applied in practical program derivations.

For real-time systems, a number of refinement calculi exist. They include one for $PL^{time}$ [76], a real-time language with CSP-like syntax with extensions for real time. [105] describes refinement of complex systems based on the Temporal Agent Model and its associated calculus.

### 5.2.2 Refinement in ITL

A refinement calculus for ITL, which is based on the TAM refinement calculus, is readily available [29]. In this technique, we distinguish between two levels of representation. The first is “abstract” and the second is “concrete”. At the first level, systems are specified at their highest level of abstraction where design and implementation issues are ignored. These issues are only considered during the transition from the abstract level to the concrete level. This transition is performed using correctness preserving refinement laws. Using such a calculus, an ITL formula could be refined into concrete code written in languages such as Ada or C.

The development technique using VisITL specifications is based on refinement. We define the refinement relation $\sqsubseteq$ in the normal way as :

$f_1 \sqsubseteq f_2 \mathrel{\widehat{=}} f_2 \supset f_1$

Since refinement is defined in terms of “implication”, we use the visual notation for “implication” to denote refinement. So, Figure 5.2 denotes refinement visually. It suggests intuitively that $f_2$ contains more details than $f_1$ because of the shaded box.

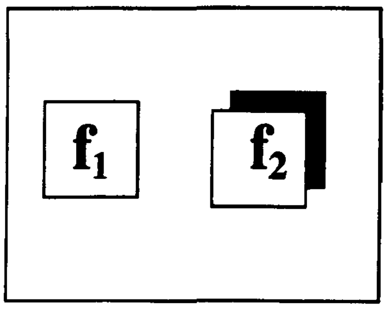


Figure 5.2: Visual refinement

The process of refinement is the same as that of program derivation. When a program is derived by the application of sound refinement rules, we can derive a proof of correctness for the statement that the program implies the specification, by tracing the steps from the program back to the specification. The process of refinement follows a common pattern for refinement calculus developments. Ordinary laws are used, first, to transform the formula until it matches the left hand side of a refinement law. Then, the matching refinement law is used to replace the formula with the right hand side part of the refinement law. Doing this successively, the formulas are replaced by program code.

This common pattern for refinement calculus developments presents us with a possibility to abstract a wide variety of users from the depth of mathematics involved in the underlying formalism.

For refining ITL specifications, one has to apply the rules defined in ITL. In particular, for refinement on VisITL specifications, we require "Visual Refinement Rules" which are defined in the following section. For each rule in VisITL, the corresponding textual ITL rule is also given for reference. We can quickly ascertain the meaning of the rule depicted visually as compared to the equivalent textual rule loaded with many notations. Also, the shaded box always quickly guides us to the meaning that what it contains is "more detailed" or "refined".

### 5.2.3 Visual Refinement Rules

In order to refine VisITL specifications visually, we define several visual refinement rules below. This repository of rules is extendable by the user with additional rules provided they are proved to be sound.

In our visual framework, we can view this as a replacement of one picture (or diagram) by another picture according to the rules of the logic. Correct refinement is thus viewed as replacing pictures with other pictures according to sound visual rules. In this way, we try to give the user a feeling that he/she is doing nothing more than a normal task which is far removed from mathematics and formal logic!

In software tools supporting this framework, annotations about the rules, i.e. an informal description of the rules, could be incorporated into the visual depiction of the rules. Such support in a visual framework will help users in feeling comfortable in a formal approach to the development of systems.

We classify the visual refinement rules into the following categories based on the purpose for which they are used.

- **Visual Refinement Rules**

  1. **General** The rules enable us to refine specifications on the same logical level.
  2. **Concrete** These rules enable us to introduce more concrete constructs during the process of refinement.

The following are some Visual Refinement Rules.

1. **General Visual Refinement Rules :**

   - **Transitivity Rule (General Visual Refinement Rule 1)**

     Textually, the rule is :

     $(ITL \sqsubseteq -1)\ (f_0 \sqsubseteq f_1)\ and (f_1 \sqsubseteq f_2)\ implies\ (f_0 \sqsubseteq f_2)$

     In VisITL, this is shown as in Figure 5.3

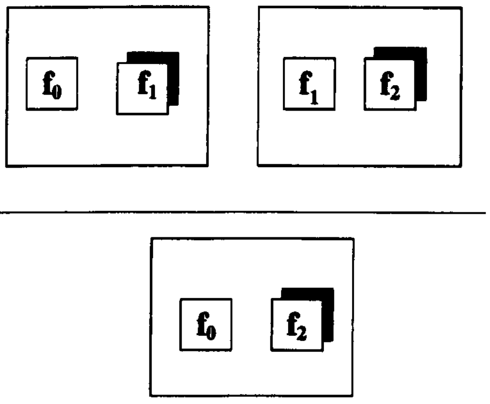


Figure 5.3: General Visual Refinement Rule 1

This rule states that if $f_1$ is a refinement of $f_0$ and $f_2$ is a refinement of $f_1$, then, $f_2$ is a refinement of $f_0$.

   - **General Visual Refinement Rule 2**

     Textually, the rule is :

$(ITL \sqsubseteq -2)\ (f_0 \sqsubseteq f_1)\ \mathit{and}(f_2 \sqsubseteq f_3)\ \mathit{implies}\ (f_0 \wedge f_2) \sqsubseteq (f_1 \wedge f_3)$

In VisITL, this is shown as in Figure 5.4

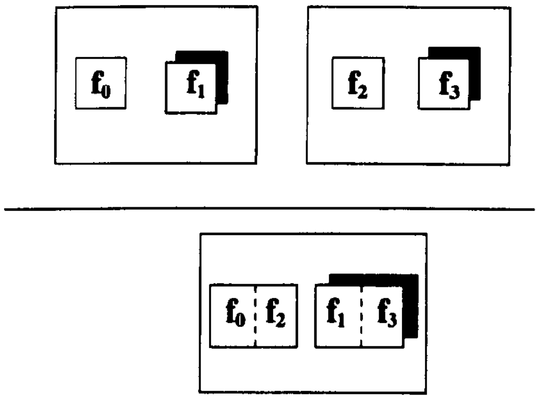


Figure 5.4: General Visual Refinement Rule 2

This rule states that if $f_0$ is refined by $f_1$ and $f_2$ is refined by $f_3$, then, the "and-composition" of $f_0$ and $f_2$ is refined by the "and-composition" of $f_1$ and $f_3$.

- **General Visual Refinement Rule 3**

  Textually, the rule is :

  $(ITL \sqsubseteq -3)\ (f_0 \sqsubseteq f_1)\ \mathit{and}(f_2 \sqsubseteq f_3)\ \mathit{implies}\ (f_0 \vee f_2) \sqsubseteq (f_1 \vee f_3)$

  In VisITL, this is shown as in Figure 5.5

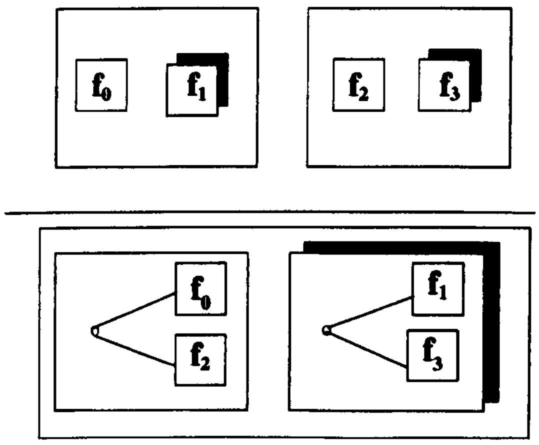


Figure 5.5: General Visual Refinement Rule 3

This rule states that if $f_0$ is refined by $f_1$ and $f_2$ is refined by $f_3$, then, the disjunction of $f_0$ and $f_2$ is refined by the disjunction of $f_1$ and $f_3$.

- **Chop Monotonicity Rule (General Visual Refinement Rule 4)**

  Textually, the rule is :

  $(ITL \sqsubseteq -4)\ (f_1 \sqsubseteq f_2)\ \textit{implies}\ (f_0 \,;\, f_1) \sqsubseteq (f_0 \,;\, f_2)$

  In VisITL, this is shown as in Figure 5.6

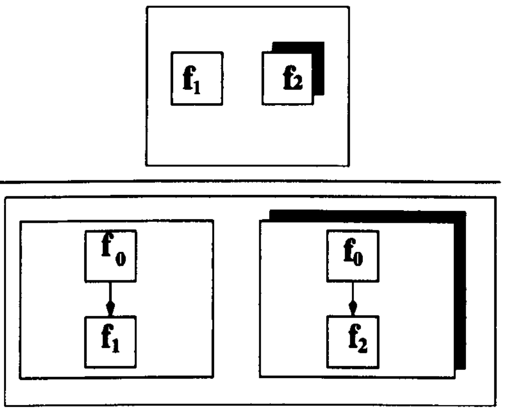


Figure 5.6: General Visual Refinement Rule 4

This rule states that if $f_1$ is refined by $f_2$, then, $f_0 \,;\, f_1$ is refined by $f_0 \,;\, f_2$.

- **General Visual Refinement Rule 5**

  Textually, the rule is :

  $(ITL \sqsubseteq -5)\ (f_0 \sqsubseteq f_1)\ \textit{implies}\ (f_0)^* \sqsubseteq (f_1)^*$

  In VisITL, this is shown as in Figure 5.7

  This rule states that if $f_0$ is refined by $f_1$, then, $(f_0)^*$ is refined by $(f_1)^*$.

2. **Concrete Visual Refinement Rules :**

- **Assignment Rule (Concrete Visual Refinement Rule 1)**

  Textually, the rule is :

  $(ITL := -1)\ \bigcirc x = exp \sqsubseteq x := exp$

  In VisITL, this is shown as in Figure 5.8

  This rule states that the ITL statement $\bigcirc x = e$ is refined by the assignment statement $x := e$.

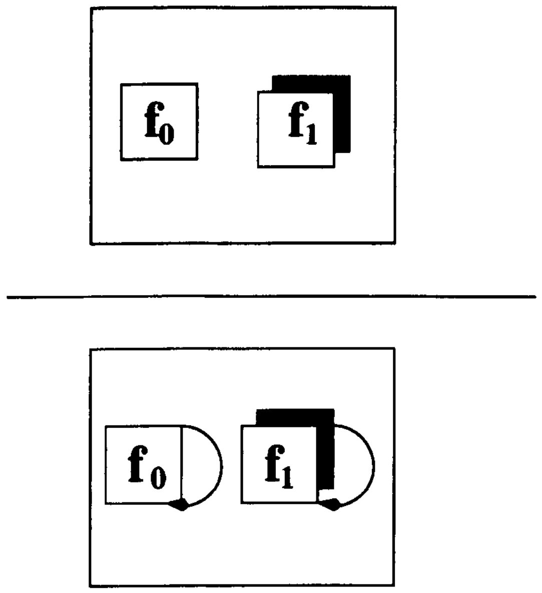


Figure 5.7: General Visual Refinement Rule 5

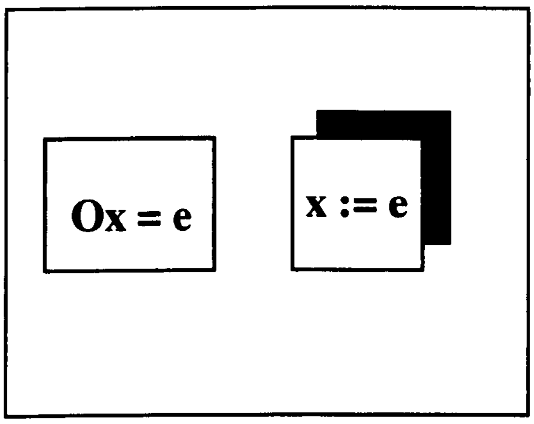


Figure 5.8: Concrete Visual Refinement Rule 1

- **If-1 (Concrete Visual Refinement Rule 2)**

  Textually, the rule is :

  $(ITL\, if\, -1)\, (f_0 \wedge f_1) \vee (f_2 \wedge f_3) \sqsubseteq if\ f_0\ then\ f_1 \Box f_2\ then\ f_3\ fi$

  In VisITL, this is shown as in Figure 5.9

  This rule states that the logic statement $(f_0 \wedge f_1) \vee (f_2 \wedge f_3)$ is refined by the "if..then" concrete construct in the shaded box which simply means that if $f_0$ holds, then, $f_1$ holds and, otherwise i.e., if $f_2$ holds, then, $f_3$ holds.

- **While-1 (Concrete Visual Refinement Rule 3)**

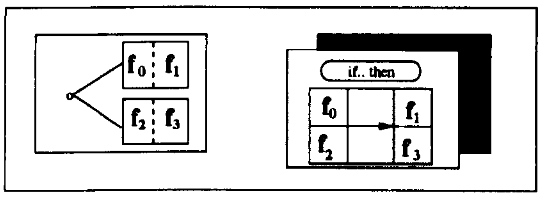


Figure 5.9: Concrete Visual Refinement Rule 2

Textually, the rule is :

$(ITL\, while\ -1)\ (f_0 \wedge f_1)^* \wedge fin(\neg f_0) \sqsubseteq while\, f_0\, do\, f_1$

In VisITL, this is shown as in Figure 5.10

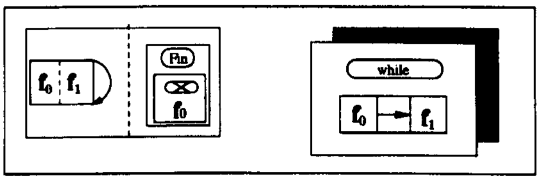


Figure 5.10: Concrete Visual Refinement Rule 3

This rule introduces a “while loop” concrete statement for a corresponding abstract VisITL formula.

- **While-2 (Concrete Visual Refinement Rule 4)**

  $(ITL\, while\ -2)\ (f_1)^* \sqsubseteq while\, true\, do\, f_1$

  In VisITL, this is shown as in Figure 5.11

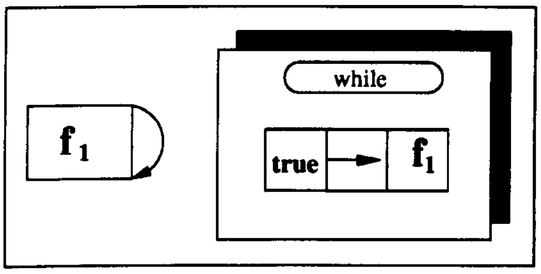


Figure 5.11: Concrete Visual Refinement Rule 4

This rule states that the logic statement $(f_1)^*$, representing a non-terminating loop in which $f_1$ is true, is refined by the concrete “while” statement in the shaded box.

- **Var-1 (Concrete Visual Refinement Rule 5)**

  Textually, the rule is :

  $(ITL\ var - 1)\ \exists x \bullet f \sqsubseteq var\ x\ in\ f$

  In VisITL, this is shown as in Figure 5.12

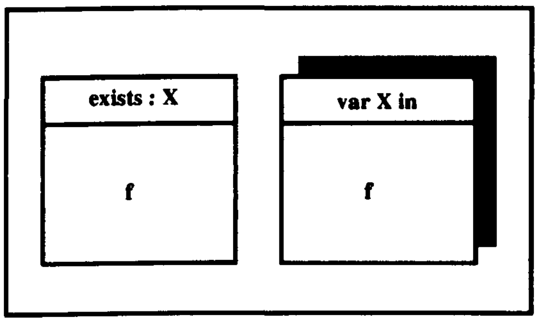


Figure 5.12: Concrete Visual Refinement Rule 5

  This rule is for the introduction of local variables.

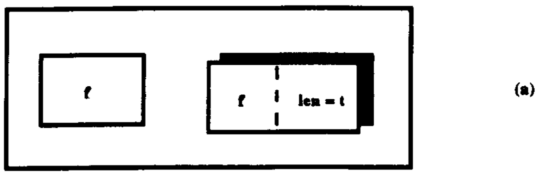
- **Execution-t (Concrete Visual Refinement Rule 6)**

  Textually, the rule could be represented in ITL as :

  $(ITL\ Execution - t)\ f \sqsubseteq f \wedge (len = t)$

  In VisITL, this is shown as in Figure 5.13 (a).

  Figure 5.13 (b) shows how it is depicted on a while loop when we choose an execution time of t steps for the body of the while loop.



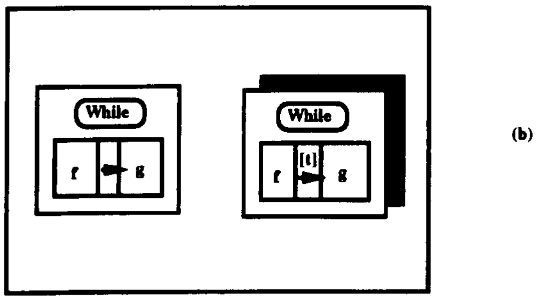


Figure 5.13: Concrete Visual Refinement Rule 6

### 5.2.4 Some Examples using Visual Refinement Rules

1. **Usage of the Transitivity Rule**

   As shown in Figure 5.14, if $0 < x < 10$ refines $x > 0$ and $5 < x < 7$ refines $0 < x < 10$, then applying the General Visual Refinement Rule i.e., the Transitivity Rule, we obtain $5 < x < 7$ as a refinement for $x > 0$.

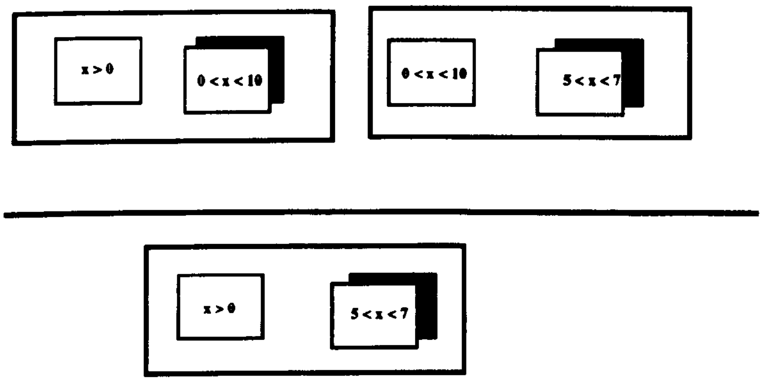


Figure 5.14: Usage of the Transitivity Rule

2. **Usage of the General Visual Refinement Rule 2**

   As shown in Figure 5.15, if $0 < x < 10$ refines $x > 0$ and $y < -5$ refines $y < 0$, then applying the General Visual Refinement Rule 2, we obtain $0 < x < 10 \wedge y < -5$ as refinement for the specification $x > 0 \wedge y < 0$.

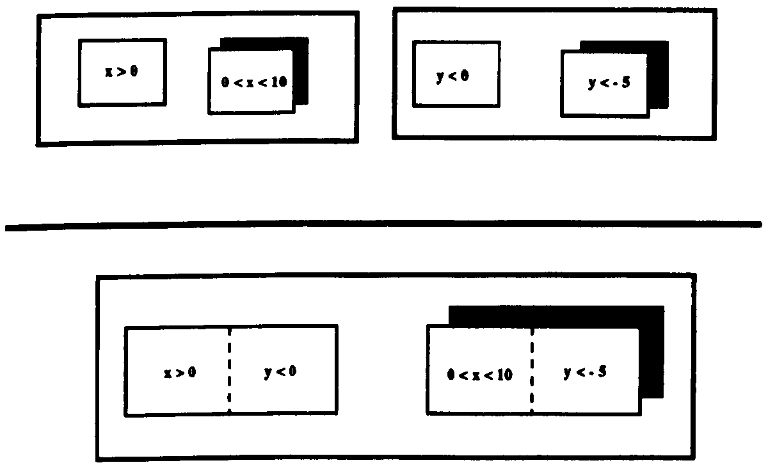


Figure 5.15: Usage of the General Visual Refinement Rule 2

3. **Usage of the Assignment Rule**

   As shown in Figure 5.16, if in the next state $x$ gets incremented by 1, then, we can introduce the assignment statement $x := x+1$ using the Assignment Rule.

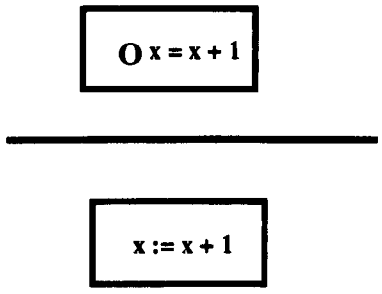


Figure 5.16: Usage of the Assignment Rule

## 5.3 Abstraction

This section will introduce abstraction and visual abstraction rules.

### 5.3.1 Introduction

Abstraction is a process of generalisation, removing restrictions, eliminating details and removing inessential information [149]. Unlike transformations, which keep the semantics unchanged, abstraction endeavours to weaken the original semantics of the system implementation. Thus abstractions cannot be applied without a clear idea of which information contained in the program refers simply to the implementation, and not to the function of the program [102]. A crucial aspect in the field of reverse engineering is this notion of abstraction since the implementation, design and specification are at different levels of abstraction.

An abstraction relation $\succeq$ is defined [103] as a function relating two agents. If an agent $f_1$ is an abstraction of $f_0$, it is written as $f_0 \succeq_R f_1 \equiv R(f_0, f_1)$ (read as $f_1$ is an abstraction of $f_0$ in respect of R) where R is defined as bending to the type of abstraction.

Abstractions can be classified as follows :

- Weakening Abstraction

  $f_0 \succeq_{WA} f_1 \equiv f_0 \supset f_1$

  Weakening abstraction refers to semantics weakening of representations (specification or code), during abstraction. In other words, if some information is taken out of the original representation and the semantics of the original representation implies the new one, then, the new representation is said to be a weakening abstraction of the original one.

- Hiding Abstraction

  $f_0 \succeq_{HA} f_1 \equiv HA(f_0, f_1)$

  means that B is a hiding abstraction of A on the condition that a part of A is hidden. This is a special case of Weakening Abstraction, where a part of the agent's data space is considered as irrelevant. Hiding Abstraction is often used to get rid of local variables and internal communication channels.

- Temporal Abstraction

  If the duration of A is denoted as T(A) and defined as $T(A) = \{x \in t \mid A \wedge len = x\}$, then,

  $f_0 \succeq_{TA} f_1 \equiv (f_0 \supset f_1) \wedge R_T(T(A), T(B))$

  is the formal definition of temporal abstraction.

  $R_T(T(A), T(B))$ is a relation between the execution times of A and B. In temporal abstraction, the execution time of the new representation can either be speeded up or slowed down compared to that of the original representation.

- Structural Abstraction

  Structural Abstraction is concerned with making structural simplification in system representation. With structural abstraction, sequential and parallel composition structures are reduced and their effects recorded in a more abstract representation. Two basic conditions which determine whether a change in system representation is a structural abstraction are firstly, whether any sequential or parallel composition is reduced in the new representation and secondly, whether the semantics of the new representation is a weakening of the original.

Structural abstraction on sequential composition, for example, is defined as follows :

$f_0 \succeq_{SA} f_1 \equiv (f_0 \supset f_1) \wedge \#seq-op(f_0) > \#seq-op(f_1)$

where $\#seq-op(f_0)$ and $\#seq-op(f_1)$ represent the number of sequential composition operators in $f_0$ and $f_1$ respectively.

- Data Abstraction

  It is a general technique by which the state space can be changed. So, one can change the original data types to higher level data types. In data abstraction, a data abstraction relation must be defined first, which maps the original data structures to new data structures and therefore the original data states to new data states. The condition of data abstraction is that the semantics of the new representation must be a weakening abstraction of the original one.

  The formal data definition of data abstraction is as follows :

  Assuming $f_0$ and $f_1$ are two representations, $r$ is a data abstraction relation :

  $r = \{(x,y) : x \in X, y \in Y, X = \{states\ of\ f_0\}, Y = \{states\ of\ f_1\}\}$.

  Therefore, $f_0$ is data abstracted to $f_1$ on relation $r$, is defined as :

  $f_0 \succeq_{DA} f_1 \equiv r(f_0) \supset f_1$

  The above definition means that $f_1$ is a data abstraction of $f_0$ on relation $r$ on the condition that, if the states of $f_0$ are mapped to those of $f_1$, then, the semantics of $f_1$ is a weakening of the mapped semantics of $f_0$.

### 5.3.2 Abstraction Rules

It has been already mentioned that in order to reverse engineer an implementation to extract a specification, one has to cross several levels of abstraction. To do this, there is a repository of abstraction rules. [103] classifies abstraction rules into two categories :

- Elementary abstraction rules

  These are rules to abstract source statements into logic formulae which may be redundant and specific. These are further classified as :

  1. Primitive abstraction rules
  2. Compound abstraction rules

- Further abstraction rules

  These rules abstract a more concise and abstract specification from the formulae through composition and semantics weakening.

It is beyond the scope of this thesis to cover the whole topic of abstraction in detail. However, many of the rules mentioned in [103] will be covered and examples in the context of how abstraction can benefit from visual ITL specifications and visual rules of abstraction given.

In this context, only the elementary abstraction rules which include primitive and compound abstraction rules will be covered.

### 5.3.3 Visual Abstraction Rules

The abstraction rules are covered below in accordance with the VisITL syntax. These rules are by no means exhaustive.

The visual abstraction rules are classified as follows :

- **Visual Abstraction Rules**

  1. **Elementary primitive** These are simple rules that enable us to abstract source code statements which may be redundant and specific into logic formulae.

2. **Elementary compound** These are less simple rules that enable us to abstract source code statements which may be redundant and specific into logic formulae.

1. **Visual Elementary Primitive Abstraction Rules :**

- **Assignment (Visual Primitive Abstraction Rule 1)**

  This rule extracts a logic formula from an assignment statement which assigns the value of expression e to variable x.

  Textually, the rule is : $x := e \succeq \bigcirc x = e$.

  This could be shown in VisITL framework as in Figure 5.17

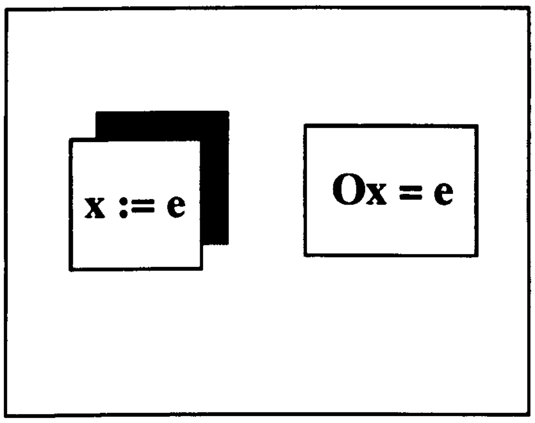


Figure 5.17: Visual Primitive Abstraction Rule 1 : Assignment

This is same as the refinement rule for introducing assignment except that the shaded box appears on the left. This change in position is just to indicate that the rule is going to be used in replacing the concrete shaded box on the left with the abstract box on the right. Otherwise, the abstraction rule formula is the same as the refinement rule formula. However, this introduction of a "new rule" which is only slightly different from the corresponding visual refinement rule could confuse the user. A better idea would be not to associate any meaning to left and right positions. As a result, one rule could be used either for refinement or for abstraction. In other words, during refinement, if the contents of the specification matched the unshaded box in the rule, then, we could use the transformation rule for refinement. During the process of abstraction, we look for a match between the VisITL specification and the contents of the shaded box in the transformation rule to determine if it could be applied for abstraction. In this manner, a visual transformation rule incorporates both refinement and abstraction. This simplifies the repository of visual rules too. I emphasise again that a visual transformation rule within the VisITL framework has one shaded box and one unshaded box to indicate the relation between the two formulae involved i.e., that the former box contains a formula that is more refined than the formula in the unshaded box. Therefore, the visual transformation rules could have these shaded and unshaded boxes in any positions relative to one another. The user simply has to think about the contents of the shaded and unshaded boxes and not worry about their positions.

- **Input Statement (Visual Primitive Abstraction Rule 2)**

  This rule extracts a logic formula from the assignment statement which reads the value in shunts (see section 4.6.7) to var y and stores the timestamp in x.

  Textually, the rule is :

  $(x,y) \leftarrow s \succeq x = \surd s \wedge y = read(s)$

  This is shown in visITL as in Figure 5.18

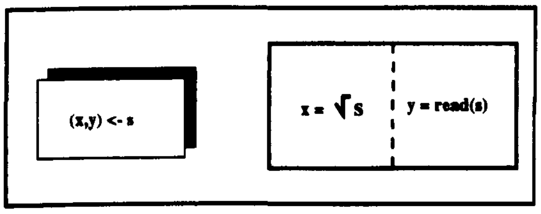


Figure 5.18: Visual Primitive Abstraction Rule 2 : Input Statement

- **Output Statement (Visual Primitive Abstraction Rule 3)**

  This rule extracts a logic formula from the output statement which writes the value of the variable or expression x to shunt s, and changes the timestamp of s to the time when the last write operation occurred.

Textually, the rule is :

$x \rightarrow s \succeq skip \wedge \bigcirc s = (\surd s + 1, x)$

This is shown in visITL as in Figure 5.19

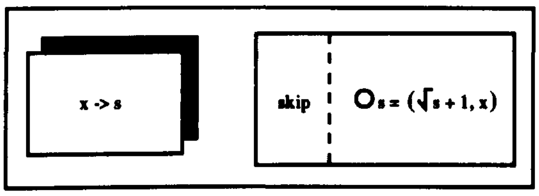


Figure 5.19: Visual Primitive Abstraction Rule 3 : Output Statement

- **Type Definition (Visual Primitive Abstraction Rule 4)**

  This rule abstracts a declaration statement declaring a variable x of type T by a logic formula which states that "there exists a variable x which has features of type T".

  Textually, the rule is :

  $x : T \succeq Type(x, T)$

  This is shown in visITL as in Figure 5.20

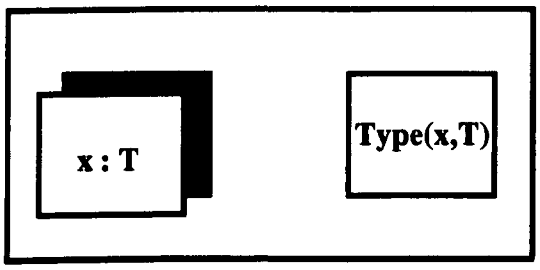


Figure 5.20: Visual Primitive Abstraction Rule 4 : Type Definition

- Delay (Visual Primitive Abstraction Rule 5)

  This abstracts a delay statement using the "len" operator in ITL.

  Textually, the rule is :

  $delay\, n \succeq len = n$

This is shown in VisITL as in Figure 5.21.

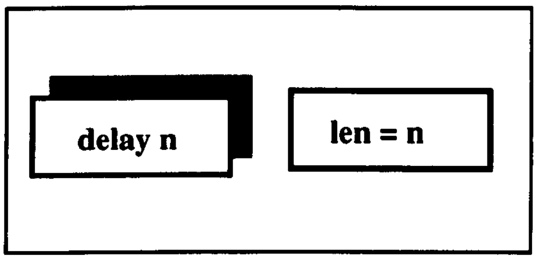


Figure 5.21: Visual Primitive Abstraction Rule 5 : Delay

- **Sequential (Visual Primitive Abstraction Rule 6)**

  This abstracts a sequential composition formula into an "and-composition". This is because the sequential composition is one of two possible refinements to an "and", the other being parallel composition.

  The rule is applicable iff $f_0$ ; $f_1$ is of the form ($f_0 \wedge$ *empty*) ;$f_1$. This condition is to be satisfied so that the abstraction is sound. So, if this soundness condition is satisfied, then, applying this abstraction would then allow a later refinement to parallel composition if desired by the user. These soundness criteria for rule application could be signalled to the user when he/she attempts to apply a rule manually. If it is an automatic application of the rule, then, this condition has to be checked by the tool automatically.

  Textually, the rule is :

  $f_0 ; f_1 \succeq f_0 \wedge f_1$

  This is shown in VisITL as in Figure 5.22.

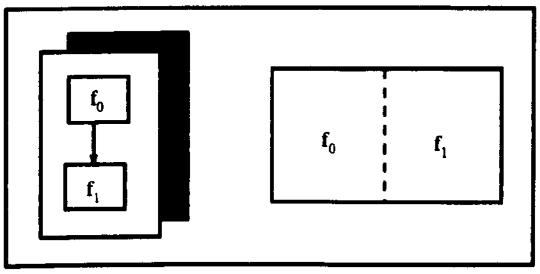


Figure 5.22: Visual Primitive Abstraction Rule 6 : Sequential

The soundness criteria for applying this abstraction rule is given by Figure 5.23.

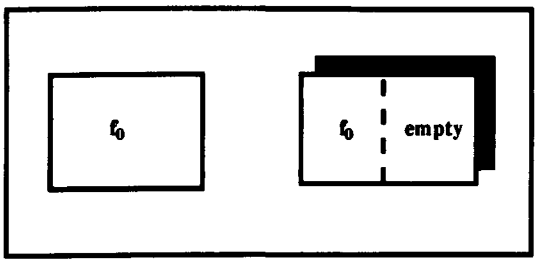


Figure 5.23: Soundness Criteria for Visual Primitive Abstraction Rule 6

2. **Visual Compound Abstraction Rules**

- Sequential Composition (Visual Compound Abstraction Rule 1)

$$\frac{f_0 \succeq f_2 \qquad f_1 \succeq f_3}{f_0 ; f_1 \succeq f_2 ; f_3}$$

If two representation fragments have a sequential relation, they can be abstracted separately, and the resultant representation composed with a sequential operator.

This is shown in visITL as in Figure 5.24

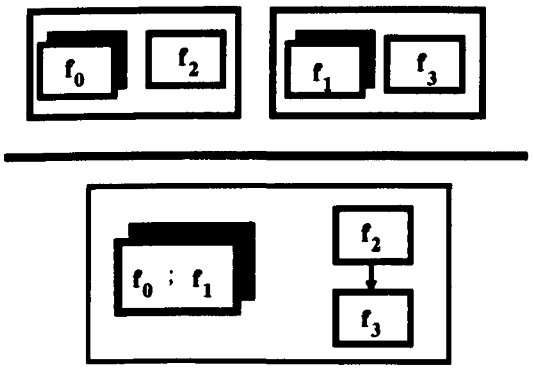


Figure 5.24: Visual Compound Abstraction Rule 1 : Sequential Composition

- **Iteration Statement (Visual Compound Abstraction Rule 2)**

This rule extracts a logic formula from an iteration statement. The iteration is mapped into "chopstar" formula in ITL while the iteration body can be abstracted separately and then joined into the chopstar structure.

Textually, the rule is :

$$\frac{f_0 \succeq f_1}{\textit{while } g \textit{ do } f_0 \succeq (g \wedge f_1)^* \wedge fin(\neg g)}$$

In VisITL, the rule is represented as in Figure 5.25.

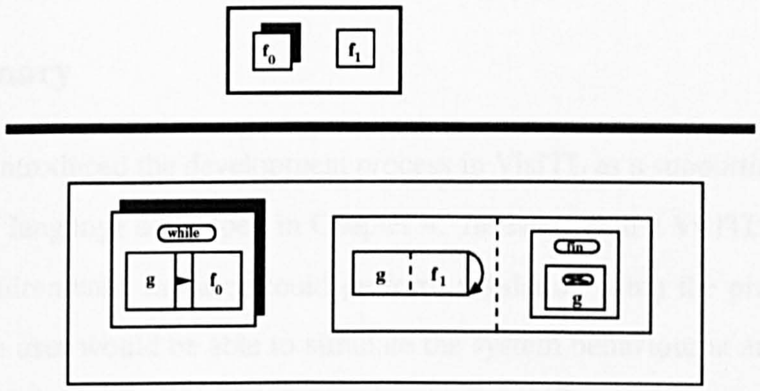


Figure 5.25: Visual Compound Abstraction Rule 3 : Iteration Statement

- **Parallel (Visual Compound Abstraction Rule 3)**

  This rule states that two concurrent or parallel representations can be abstracted separately and the results composed through the conjunction operator.

  Textually, the rule is :

  $$\frac{f_0 \succeq f_2, f_1 \succeq f_3}{f_0 \,||\, f_1 \succeq (f_2 \wedge f_3)}$$

  In VisITL, the rule is represented as :

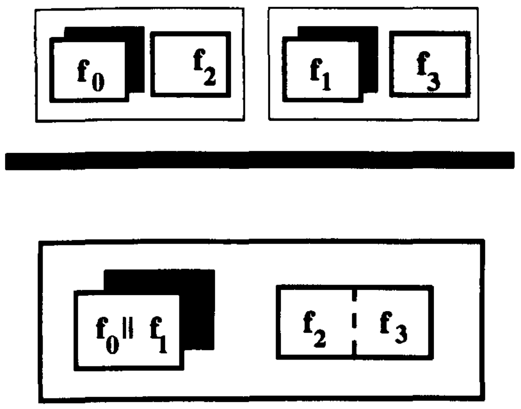


Figure 5.26: Visual Compound Abstraction Rule 3 : Parallel

## 5.4 Summary

This chapter introduced the development process in VisITL as a supporting framework to the VisITL language developed in Chapter 4. Initially, with a VisITL specification capturing requirements, the user could perform validation using the proof system of ITL. Also, the user would be able to simulate the system behaviour at any abstraction level using Tempura. To allow the user to seamlessly go forward from the abstract VisITL specification, sound visual refinement rules in the framework are needed. In other words, just having a VisITL specification capturing requirements is not enough. To derive an implementation i.e., forward engineer the specification, visual refinement rules are needed in the framework. Similarly, in order for the user to perform reverse engineering [103], i.e. derive a high-level specification from an existing implementation, we needed to provide sound abstraction rules in the VisITL framework. For refinement of VisITL specifications in a visual framework, visual refinement rules were introduced with some illustrative examples. This framework now enables users to simply use appropriate visual rules incorporated into the repository and refine VisITL specifications. Also, abstraction rules were introduced to facilitate the re-engineering process in a visual framework. In the next chapter, simple examples and case studies in VisITL will be seen.

# Chapter 6

# Case Studies in VisITL

In chapter 4, a simple notation for VisITL was introduced along with examples in VisITL. In this chapter, we shall look at a case-study in VisITL specification considering the development technique of refinement introduced in the previous chapter. In addition, we shall, briefly, consider an example where abstraction is performed within the visual framework to obtain an abstract VisITL specification.

## 6.1 A Robot Control System

This case study is an application involving multiple processes [26] for a robot control system. An informal description of the system is given below :

*"The tele-operated robot is a tracked device which was originally developed for military use. The carriage can easily traverse over rough terrain. The vehicle schematic is shown in Figure 6.1. The vehicle has on-board a manipulator arm that has three degrees of freedom controlled by hydraulic actuators. The electric drive motors, manipulator actuators and on-board lamps are controlled manually by the operator via a control box that is linked to the vehicle. Currently one controller caters for the main control, one for the infrared sensor interfacing and processing, and a third for the on-board camera control.*

*The actual vehicle is driven by two motors, left and right, indicated as L and R in*

*Figure 6.1. Both of these motors can move forwards and in reverse. The vehicle is steered by moving one motor faster than the other.*

*From a control point of view, commands are issued to the motors via a operator joystick (L and R of the operator console in Figure 6.1 which issues integer values in the range 0...127 for forward motion and 0...-128 for reverse motion. It is possible to drive only one motor at a time, in such a case, the robot will turn. The speed of the motors is directly proportional to the value written to them.*

*The robot is equipped with 8 infra red sensors. These return an integer value in the range 0...255 depending on whether an obstacle is present or not. 0 indicates no obstacle, 255 indicates obstacle very near. We normally operate with a threshold of around 100, above which we take notice of the sensor readings, i.e., an obstacle is of interest. At this point, reactive control takes over from the manual control by moving the vehicle away from the obstacle until the 100 threshold is not set. The sensor positions are as follows : N, NE, E, SE, S, SW, W and NW, covering the body of the robot as shown in Figure 6.1."*

The following presents the specification and the design of the driving part of the robot control system.

The specification can be divided into 3 parts, one each for motor control, infra-red control and operator control. The specification for the robot control system is given by Figure 6.2. The ITL formula for it is :

ROBOT CONTROL SYSTEM $\equiv$

(MOTOR CONTROL SYSTEM) $\wedge$ (INFRA-RED CONTROL SYSTEM) $\wedge$ (OPERATOR CONTROL SYSTEM)

We can, now, separately consider each of the three components for specification and refinement.

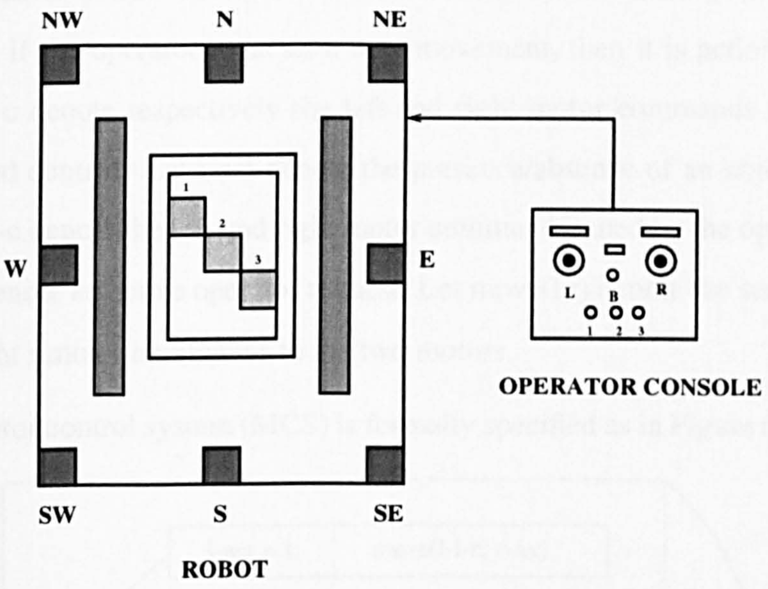


Figure 6.1: A Schematic of the Robot Control System

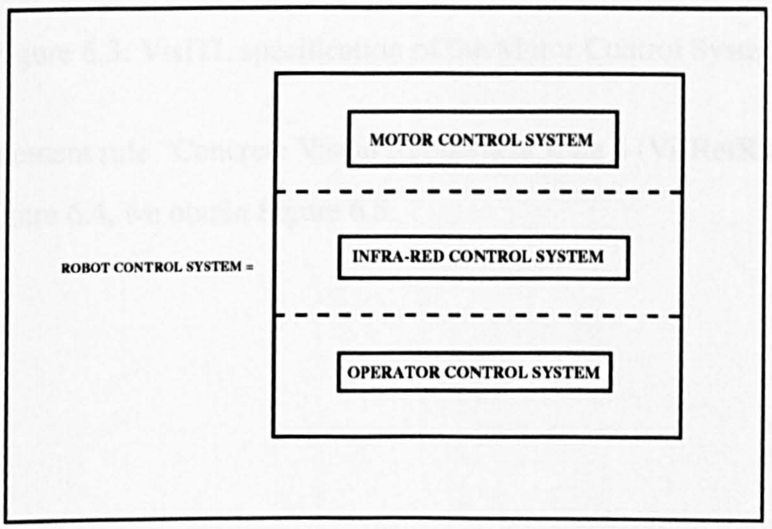


Figure 6.2: VisITL specification of the Robot Control System

1. **Motor Control :**

   If the sensor detects an object, then the control system takes over control ; otherwise, if the operator requests a new movement, then it is actioned. Let l-i-c and r-i-c denote respectively the left and right motor commands issued by the infra-red control. Let i-act denote the presence/absence of an object. Let l-o-c and r-o-c denote the left and right motor command issued by the operator and let o-act denote an active operator request. Let move(l,r) denote the sending of left l and right r motor commands to the two motors.

   The motor control system (MCS) is formally specified as in Figure 6.3. Applying

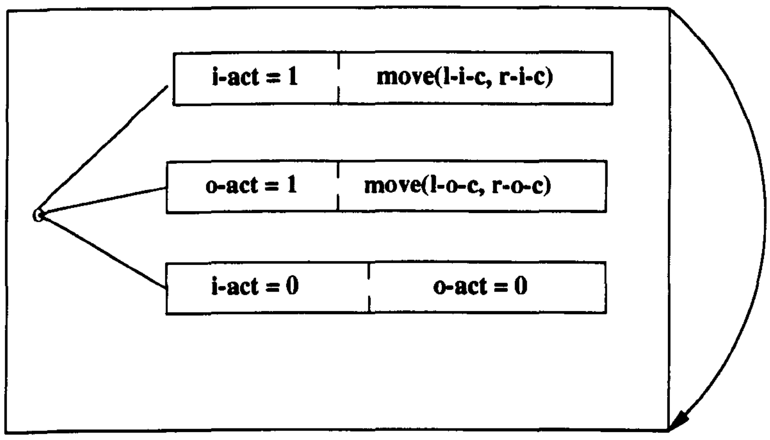


Figure 6.3: VisITL specification of the Motor Control System

the refinement rule "Concrete Visual Refinement Rule 4 (VisRefRule : While2)" as in Figure 6.4, we obtain Figure 6.5.

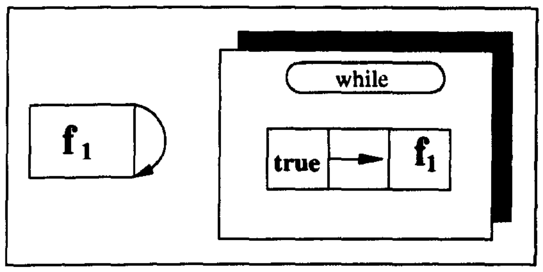


Figure 6.4: The While-2 Refinement Rule

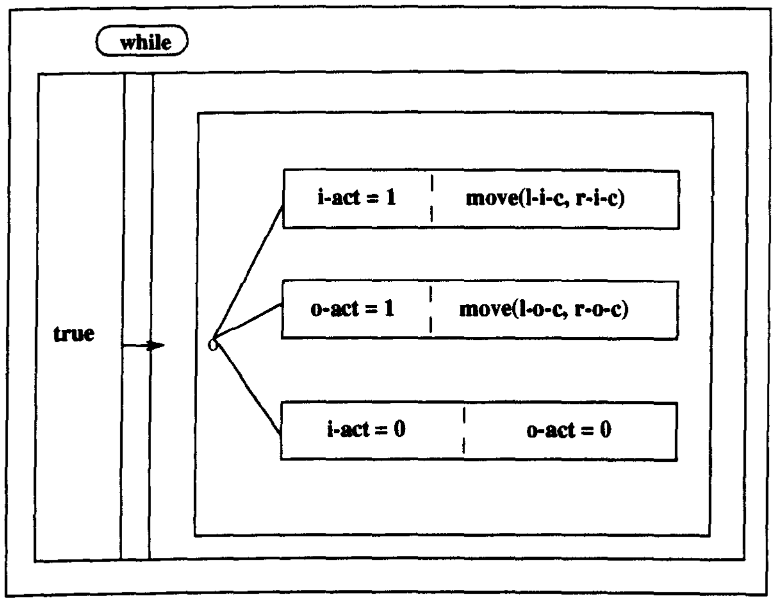


Figure 6.5: VisITL MCS specification after Applying the While-2 Refinement Rule

Introducing the "If then" with the "Concrete Visual Refinement Rule 2 (VisRefRule : If1" as shown in Figure 6.6, we obtain Figure 6.7. Now, the decision to

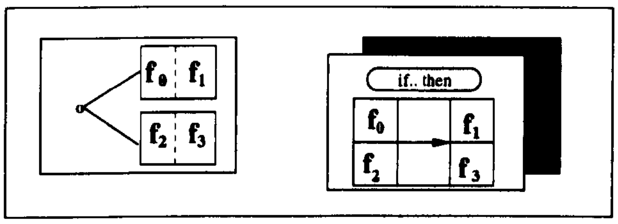


Figure 6.6: The If-1 Refinement Rule

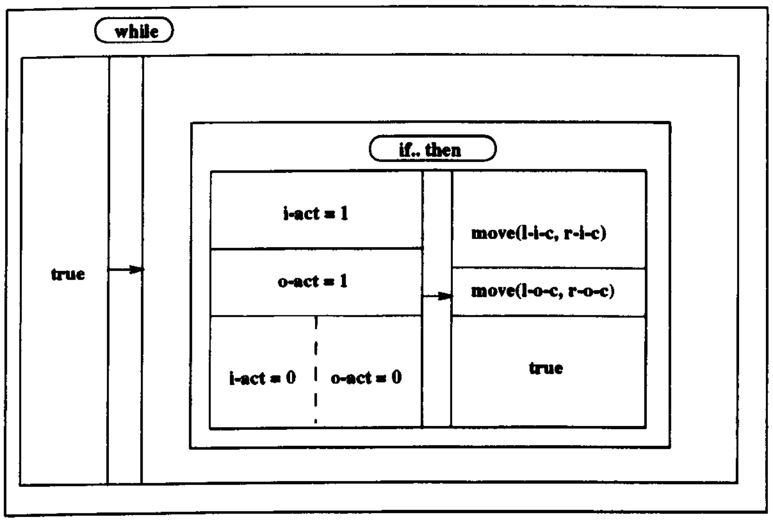


Figure 6.7: VisITL MCS specification after Applying the If-1 Refinement Rule

be taken for the issuing of the move commands is not deterministic in case both infrared and operator are active. Let us say that we want to design the control in such a way that the operator is given precedence. We can state this in the form of a design decision rule as shown in Figure 6.8. Applying this design decision rule, we obtain the specification in Figure 6.9. The design rules could be implemented as general rules as in Figure 6.10 and during the development process, the parameters 'f' and 'g' could be mapped to the parameters desired. This is a way forward in aiding the user with Visual design decision rules.

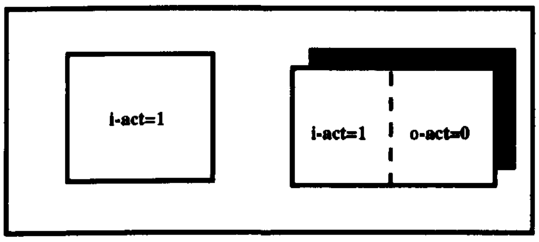


Figure 6.8: Refinement of i-act due to our Design Decision

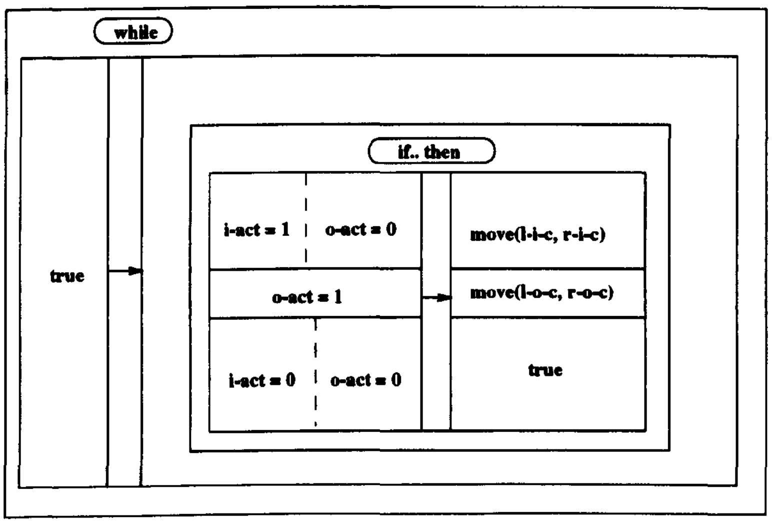


Figure 6.9: VisITL MCS specification after Applying the Design Decision

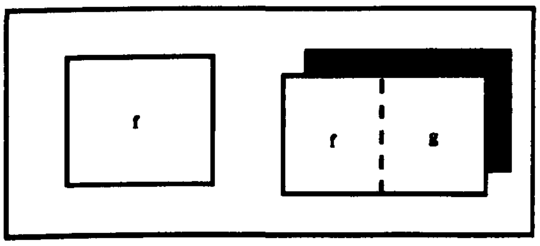


Figure 6.10: Visual Refinement Rule - General Design Decision 1

Now, applying the Concrete Visual Refinement Rule 6 (Execution-t Rule) for choosing a specific execution time of 't' steps for the body of the while loop, we obtain the specification in Figure 6.11.

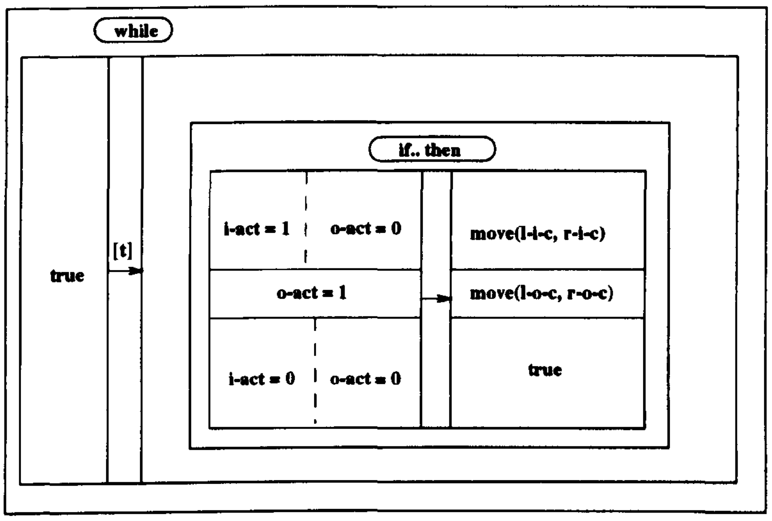


Figure 6.11: VisITL MCS specification for an Execution Time of t Steps

The following is the refinement in textual ITL for comparison in sect. 6.2.

$MOTOR\,CONTROL\,SYSTEM\ =$

$($

$((i-act\ =\ 1)\ \wedge\ move(l-i-c, r-i-c))\ \vee$

$((o-act\ =\ 1)\ \wedge\ move(l-o-c, r-o-c))\ \vee$

$((i-act\ =\ 0)\ \wedge\ (o-act\ =\ 0))$

$)*$

Applying the ITL $\sqsubseteq$ while-2, we obtain :

$while\ true\ ($

$((i-act\ =\ 1)\ \wedge\ move(l-i-c, r-i-c))\ \vee$

$((o-act\ =\ 1)\ \wedge\ move(l-o-c, r-o-c))\ \vee$

$((i-act\ =\ 0)\ \wedge\ (o-act\ =\ 0))$

$)$

Strengthening the guard of the infra-red move as before, by applying the

$\mathit{design\ decision\ rule} : (i-act = 1) \sqsubseteq (i-act = 1) \wedge (o-act = 0)$,

we obtain

$\mathit{while\ true}$ (

$((i-act = 1) \wedge (o-act = 0) \wedge move(l-i-c, r-i-c)) \vee$

$((o-act = 1) \wedge move(l-o-c, r-o-c)) \vee$

$((i-act = 0) \wedge (o-act = 0))$

)

Applying the ITL $\sqsubseteq$ if-1 rule, we obtain :

$\mathit{while\ true}$ (

$\mathit{if}\ ((i-act = 1) \wedge (o-act = 0))\ \mathit{then}\ move(l-i-c, r-i-c)$

$\square\ (o-act = 1)\ \mathit{then}\ move(l-o-c, r-o-c)$

$\square\ ((i-act = 0) \wedge (o-act = 0))\ \mathit{then\ true}$ )

$\mathit{fi}$)

2. **Infra-red Control :**

The sensors are to be read and for each sensor reading that is greater than the threshold of 100, the motor commands are to be adjusted accordingly.

Let $ir-c(i)$ denote the sensor $i$(N: i=0, NE: i=1, E: i=2, SE : i=3, S: i=4, SW : i=5, W: i=6, NW : i=7). Let $mvl(i)$ and $mvr(i)$ denote respectively the left and right steering commands corresponding to sensor $i$. Also, let us denote the sum of all $mvl(i)$ values by $sum(mvl(i))$ and the sum of all $mvr(i)$ by $sum(mvr(i))$. In our example, the sum function could be defined as :

$sum(X) = (\mathit{if}\ ir-c(0) > 100\ \mathit{then}\ X(0)\ \mathit{else}\ 0) +$

$(\mathit{if}\ ir-c(1) > 100\ \mathit{then}\ X(1)\ \mathit{else}\ 0) +$

$(\mathit{if}\ ir-c(2) > 100\ \mathit{then}\ X(2)\ \mathit{else}\ 0) +$

$(......) +$

$(\mathit{if}\ ir-c(7) > 100\ \mathit{then}\ X(7)\ \mathit{else}\ 0)$

This function adds up the values corresponding to the 8 sensors. Each sensor contributes a value only if it is above the given threshold of 100.

Here, we note that the function implementation is not given a visual representation. However, we could extend our visual framework with a graphical way of depicting functions. In this context, it is interesting to note how it could be done similar to the graphical functions within the stateflow formalism [143]. Such a graphical function integrated into the VisITL framework would look like in Figure 6.12. The condition/the guard is put in square brackets and the corresponding

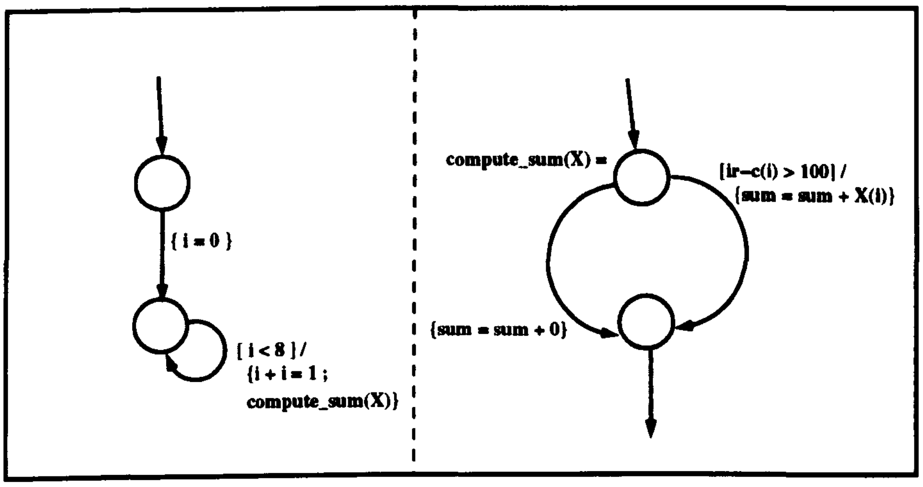


Figure 6.12: A Graphical Depiction for Computing the Sum of Sensor Contributions

action, if any, in flower brackets.

The specification for ICS is given formally by the VisITL specification in Figure 6.13. Using rule Vis-while2, we obtain the specification in Figure 6.14.

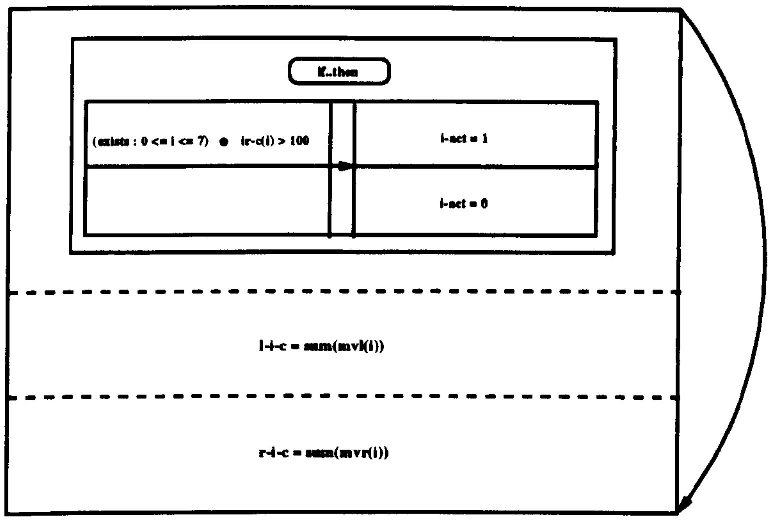


Figure 6.13: VisITL Specification for the Infra-red Control System

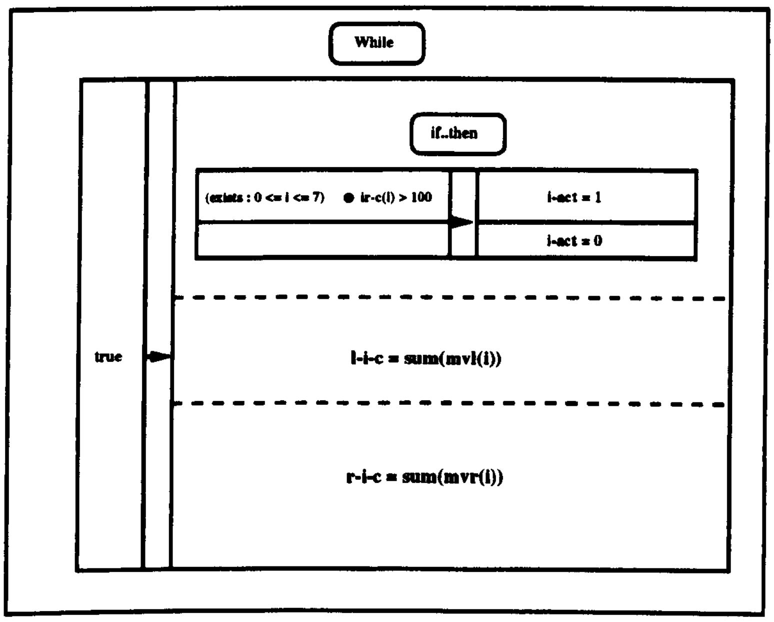


Figure 6.14: VisITL ICS Specification after Applying Vis-while2

Applying the Concrete Visual Refinement Rule 6 (Execution-t Rule) for choosing an execution time of "t" steps for the body of the while loop would result in Figure 6.15. So, for an execution time of t, the specification can be shown in refined form

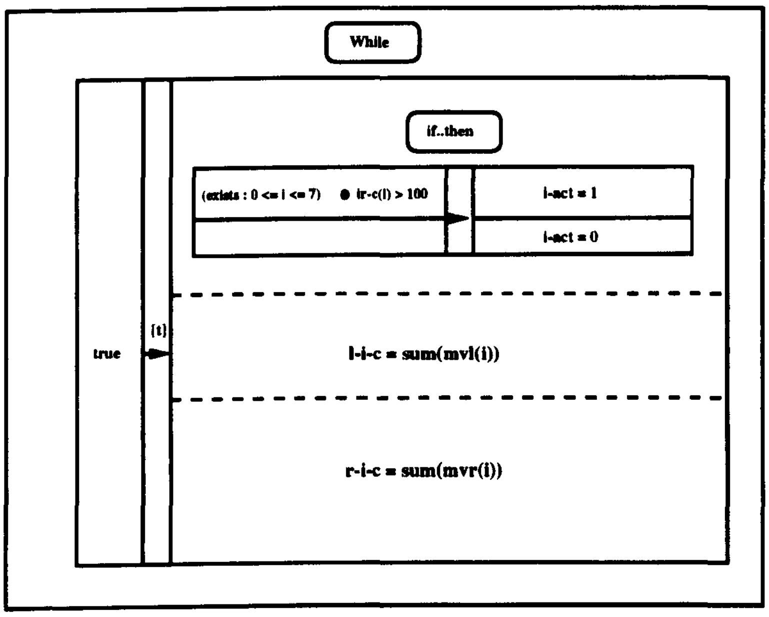


Figure 6.15: VisITL ICS Specification for an Execution Time of t Steps

as in Figures 6.16 and 6.17. This will sample the sensors at the beginning of the execution interval, set $i - act$ accordingly and disable $i - act$ during the rest of the execution interval.

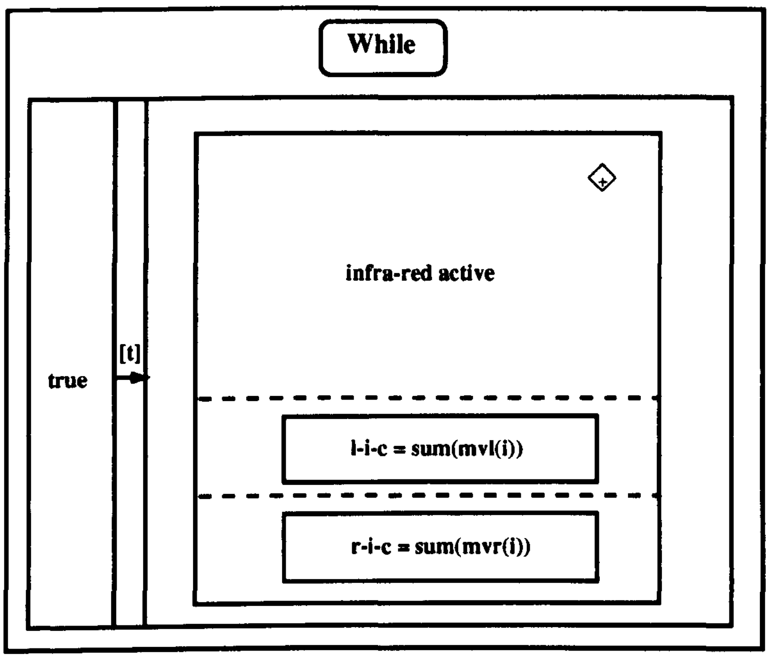


Figure 6.16: The VisITL ICS Specification with Zoom-in for infra-red active

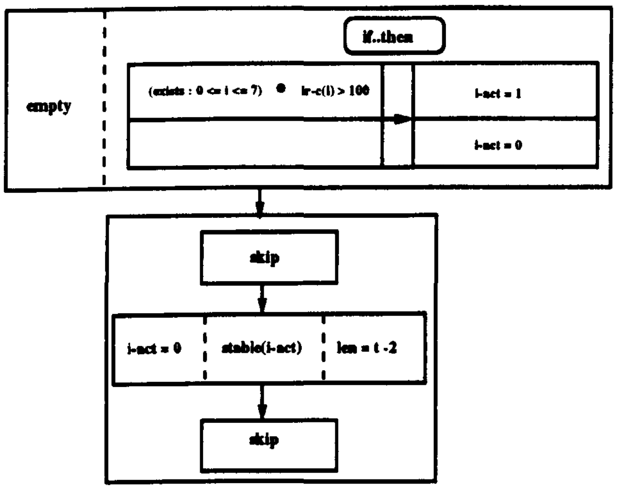


Figure 6.17: The Zoomed-in infra-red active for Execution Time of t

If we choose an execution time of 1 i.e., one time step, then, the specification would be represented as in Figure 6.18.

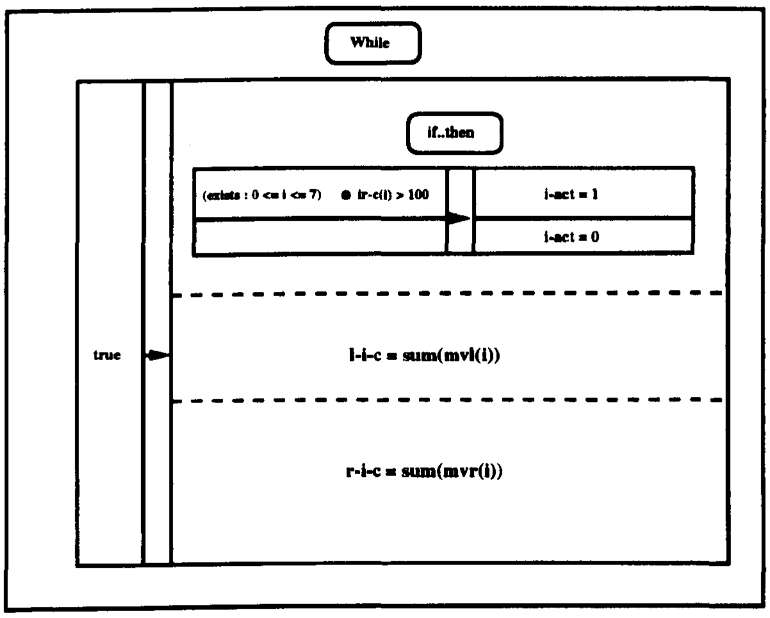


Figure 6.18: Final Refined ICS for the Special Case of Unit Execution Time

3. **Operator Control :**

   If the operator requests some changes, then, these are to be processed.

   Let $ll-o-c$ and $lr-o-c$ denote respectively the last left and last right steering commands received from the operator.

   The specification of the OCS is given by Figure 6.19. Using rule Vis-var1, we can introduce local variables and obtain the specification Figure 6.20. Using rule Vis-while2, we can obtain the specification in Figure 6.21. The assignment statements can be introduced using Visual Refinement Rule 6 to obtain Figure 6.22.

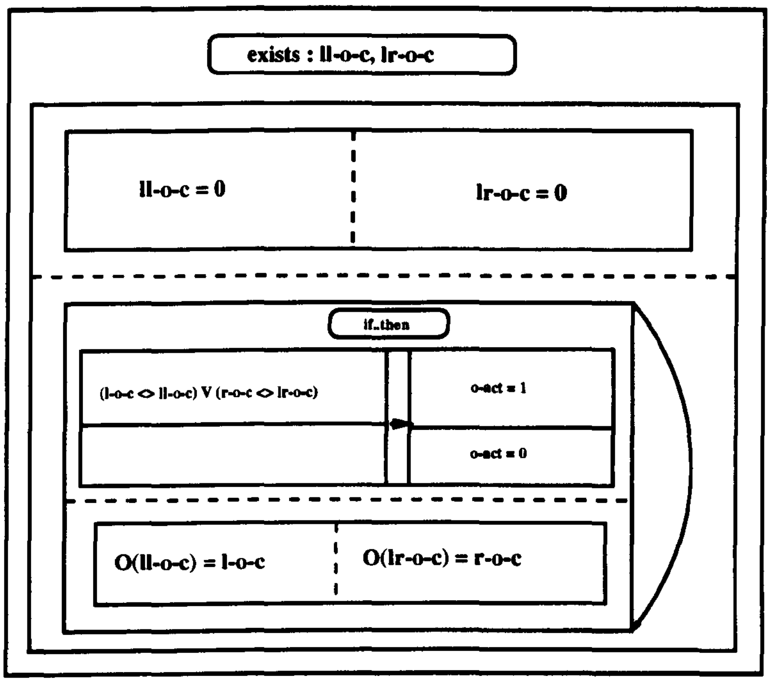


Figure 6.19: VisITL Operator Control System Specification

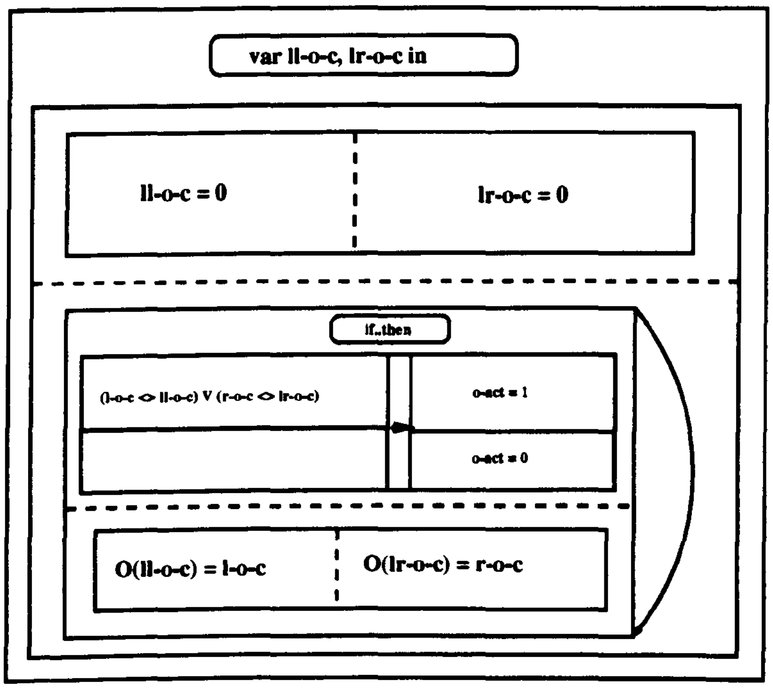


Figure 6.20: VisITL OCS Specification after Applying Vis-var1

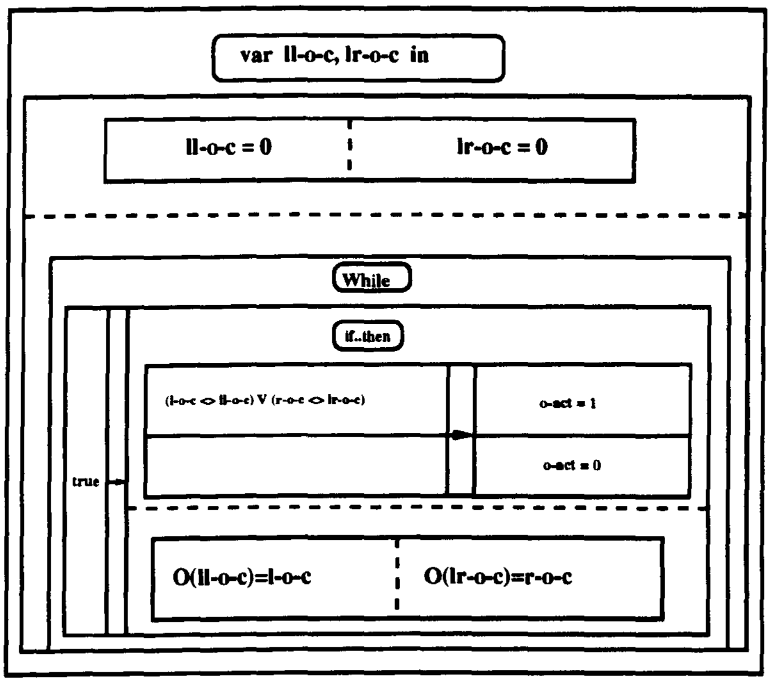


Figure 6.21: VisITL OCS Specification after Applying Vis-while2

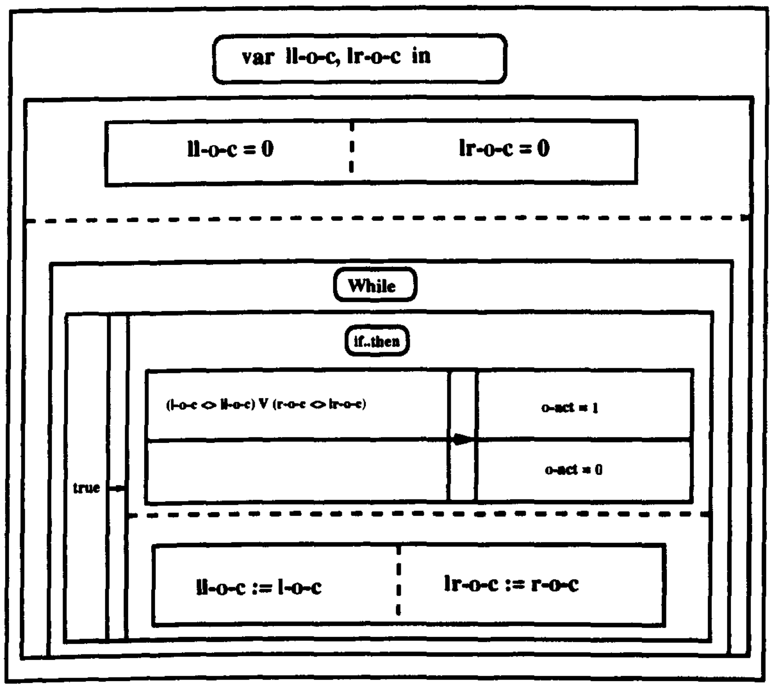


Figure 6.22: VisITL OCS Specification after Introducing the Assignment Statements

Finally, applying the Execution-t Rule for choosing an execution time of "t" steps for the body of the while loop results in the concrete code in Figure 6.23. So,

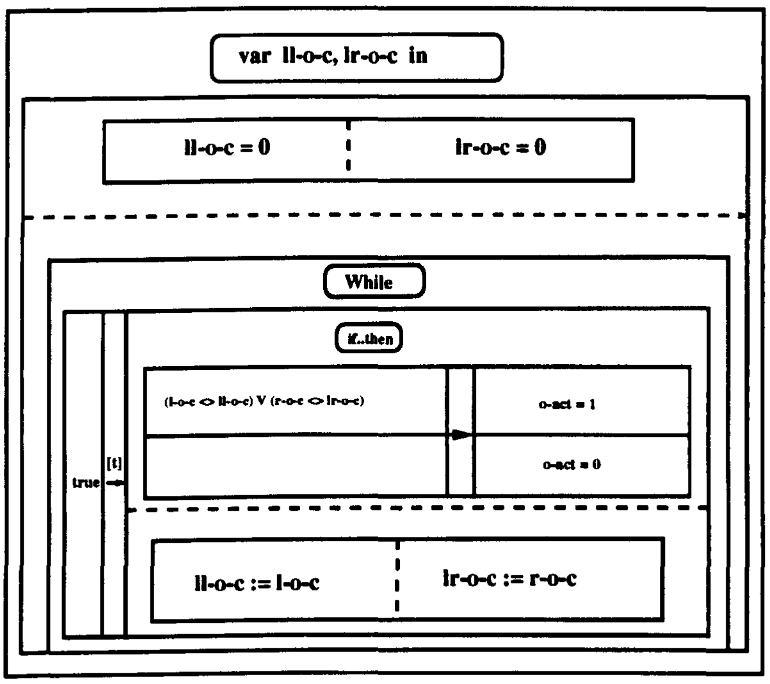


Figure 6.23: VisITL OCS Specification for an Execution Time of t Steps

for an execution time of t, the specification can be shown in refined form as in Figures 6.24 and 6.25. This will sample the operator signals at the beginning of the execution interval, set $o - act$ accordingly and disable $o - act$ during the rest of the execution interval.

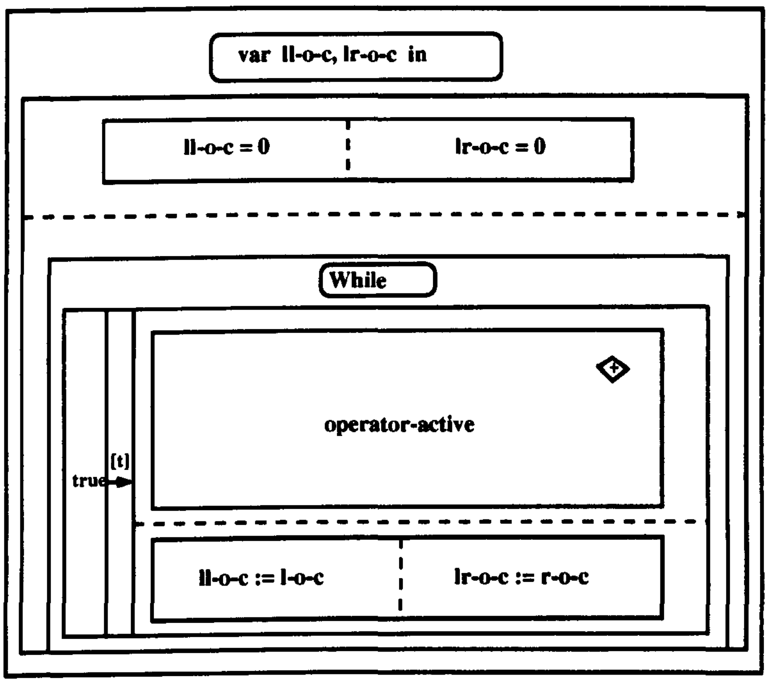


Figure 6.24: VisITL OCS Specification with Zoom-in for operator-active

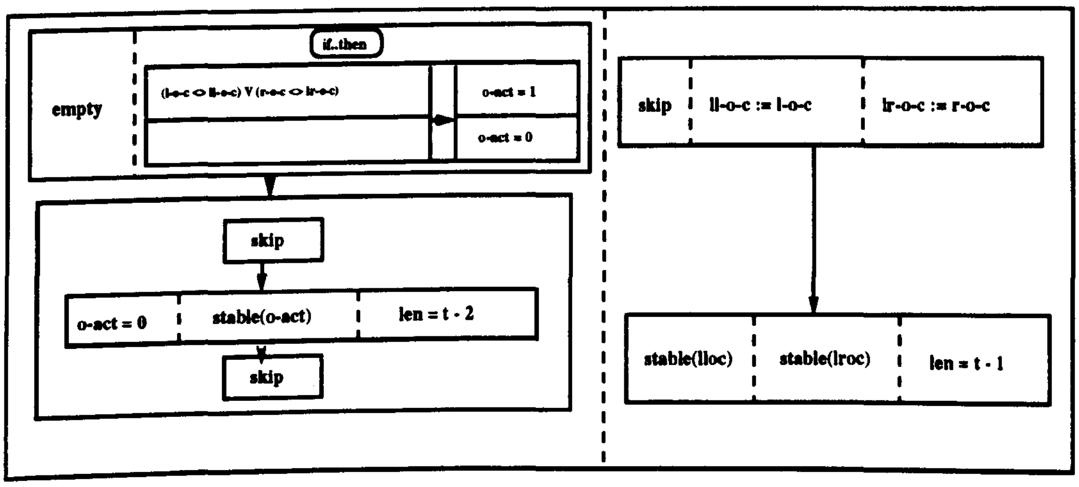


Figure 6.25: Operator-active for an Execution Time of t Steps

If we choose an execution time of 1 i.e., one time step, then, the specification would be represented as in Figure 6.26.

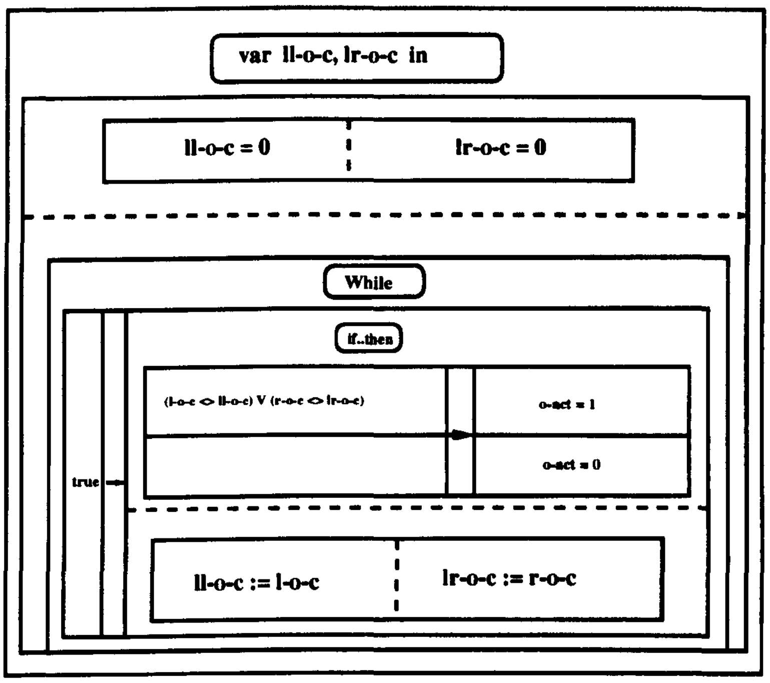


Figure 6.26: Final Refined OCS for the Special Case of Unit Execution Time

## 6.2 Comments on the Two Approaches

It is easy to see that the transformations on purely textual ITL specifications are not that easy to keep track of despite good indenting. There are not many visual cues to rely on, despite good indentation of the textual specification. This is a crucial point in dealing with even larger specifications on which we might end up applying numerous transformations. This adds further strain, especially on a user who is not dealing with mathematical specifications on a regular basis. Even formalists would be hard-pressed to keep track of the development process especially when automated tools are being used on ITL specifications. One advantage however with textual specifications is that they are very concise. The refinement of MCS in textual ITL fits on a single page unlike the refinement in VisITL which takes several pages. Also, one could manage to write down textual specifications using "pen and paper" despite their inevitable dependency on "not-so-easy-to write" mathematical symbols whereas visual specifications are more suited for use with an appropriate software tool. In this context, it is interesting to mention that the ancient Egyptian Hieroglyphic script is a writing system that employs characters in the form of pictures and dates back to the end of 4th millennium B.C.. In [19] the interested reader has a nice "self-learning" reference to learn how to write using "complicated pictures" without being too much of an artist!

In contrast, in the visual framework, it was easy to see what happened to the specification on application of a refinement rule. It was possible to see that the contents of boxes were being moved around as depicted by the visual refinement rule adding to the confidence of the user that the process was progressing as specified. The user gains better control of the formal development process with such visual cues.

Hence, sacrificing the conciseness of textual specifications for wider accessibility of the formal approach is a worthy compromise indeed. In any case, one could always translate a VisITL specification to its concise ITL form for the experienced formalist.

Another interesting point to mention here is that a visual specification provides a possible benefit in resolving non-determinism. In the above example, we removed the non-determinism that existed by applying a design decision rule to strengthen the guard of the infra-red move. However, we could incorporate some geometry rules in order to automatically resolve non-determinisms. Maybe, as a default, we could apply some geometry rules to resolve non-determinism which could be over-ridden by the user, if required, through appropriate design rules.

Let us consider the specification for motor control system as shown in Figure 6.27. Here, the guards are put on the outgoing lines of the visual "Or" construct. This is a possible new special notation for the "or". We could have a default rule to consider the guards in a particular order. If we consider the guards from a 12 'o' clock position on the node at which the "or" lines originate in a clockwise manner as depicted in Figure 6.29, then, the guard $i-act$ gets the first priority. In this case, we ignore the guards $o-act$ and $(\neg\, i-act \;\wedge\; \neg\, o-act)$. If $i-act$ is not satisfied, then, we check $o-act$ and so on. In this way, we assign the guard $i-act$ the highest priority. This approach is

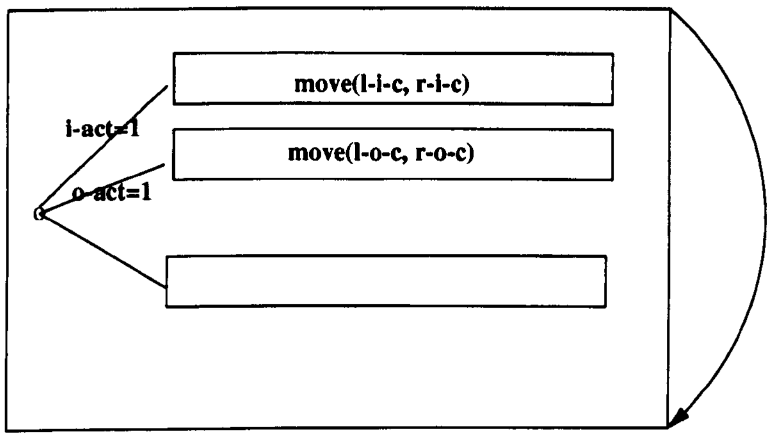


Figure 6.27: The i-act Component of the Visual-Or has Higher Priority.

similar to the approach used within the stateflow visual formalism [143] for resolving non-determinism based on geometry.

For the operator to be able to override infrared control as we did using a design decision rule, we have to re-order the boxes as shown in Figure 6.28.

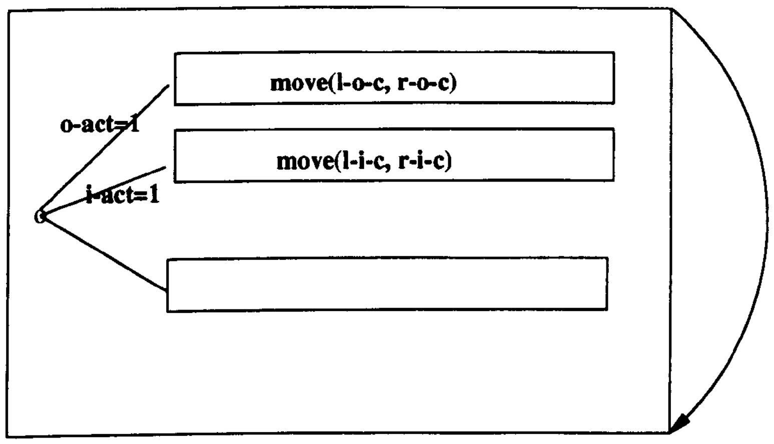


Figure 6.28: The Box Positions are now changed to give the Operator Higher Priority.

Alternately, it may be a good idea not to attach any meaning to the geometry so that non-determinism might be resolved only by the user using some design decision rule.

In my examples, I have not attached any meaning to the geometry of the "or" lines in the Visual "Or" construct.

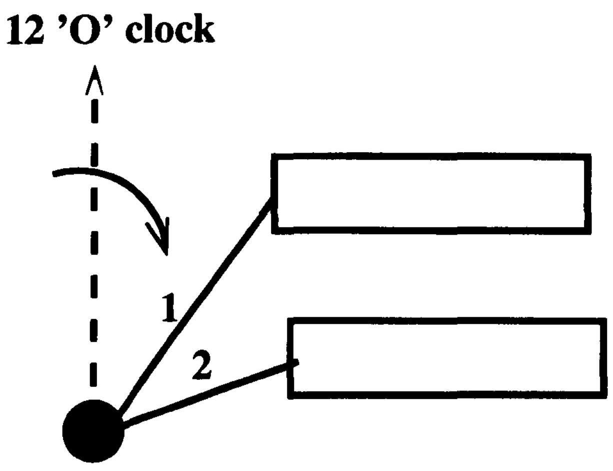


Figure 6.29: Guard rule using Geometry for Resolving Non-determinism

## 6.3 A Mine Pump Control System

In the previous example, a case study in VisITL involving refinement was considered. In the re-engineering of systems, extracting an ITL specification from an existing implementation becomes important. For this purpose, the techniques of abstraction in ITL described in [102, 103] are of utmost importance. The following case study demonstrates how abstraction in ITL can be incorporated in our visual framework.

This is a commonly occurring case study in formal methods literature. An elaborate description of the problem can be found in [24]. An informal specification of the problem is reproduced from [91] below with a schematic diagram shown in Figure 6.30: *"Water percolating into a mine is collected in a sump to be pumped out of the mine. The water level sensors D and E detect when water is above a high and low level respectively. A pump controller switches the pump on when the water reaches the high water level and off when it goes below the low water level. If, due to a failure of the pump, the water cannot be pumped out, the mine must be evacuated within one hour.*

*The mine has other sensors (A, B, C) to monitor the carbon monoxide, methane and airflow levels. An alarm must be raised and the operator informed within one second of any of these levels becoming critical so that the mine can be evacuated within one hour. To avoid the risk of explosion, the pump must be operated only when the methane*

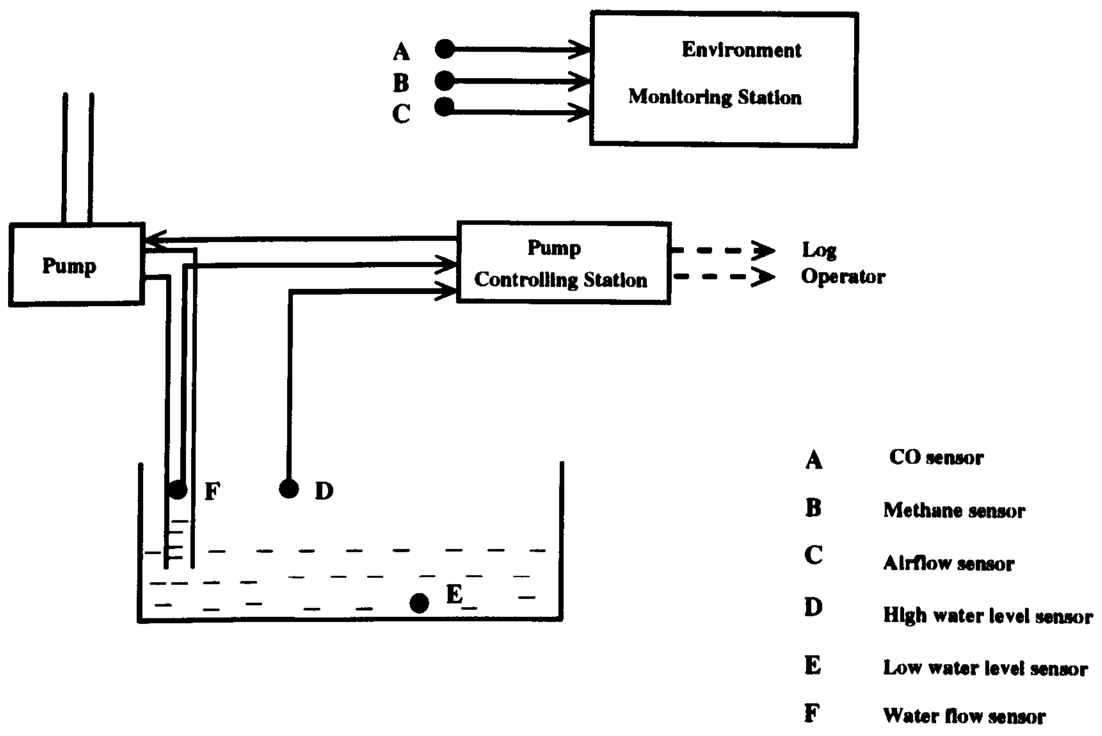


Figure 6.30: A Mine Pump Control System

*level is below a critical level.*

*Human operators can also control the operation of the pump, but within limits. An operator can switch the pump on or off if the water is between the low and high water levels. A special operator, the supervisor, can switch the pump on or off without this restriction. In all cases, the methane level must be below its critical level if the pump is to be operated. Readings from the sensors, and a record of the operation of the pump, must be logged for later analysis.*

The main safety requirement is that the pump should not be operated when the level of methane gas in the mine reaches a high value due to the risk of explosion.

For demonstrating an application in abstraction, an implementation of the above requirements in ADA which was subsequently translated into a Common Structural Language (CSL)[1] in [103] is chosen as the basis. This case study demonstrates abstraction using VisITL specifications.

[1]CSL was developed [103] to enrich statements in Time Guarded Command Language and make Reengineering Wide Spectrum Language (RWSL) compatible to WSL in Maintainer's Assistant tool

One module called the "pump module" is selected from the CSL code and translated it into Tempura code. The Tempura code is given below :

```
define Motor-unsafe() = {
    if (Motor-status = On) {
           Sw := Off ;
           Motor-status := Off ;
           format(''motor-stopped \n '')
    } ;
    Motor-condition := Disabled ;
    format(''motor-unsafe \n'')
} and
define Motor-safe() = {
    if (Motor-status = Off) {
           Sw := On ;
           Motor-status := On ;
           format(''motor-started \n'')
    } ;
    Motor-condition := Enabled ;
    format(''motor-safe \n'')
} and
define set-pump(Pump-status) = {
    if (Pump-status = On) {
       if (Motor-status = Off) {
          if (Motor-condition = Disabled) {
             format(''pump-not-safe \n'')
          } ;
          if (Ch4-status = Motor-safe) {
             Motor-status := On ;
             Sw := On ;
```

```
            format(''motor-started \n'')
         }
         else format(''pump-not-safe \n'')
      } ;
      else if (Motor-status = On) {
               Motor-status := Off ;
               if (Motor-condition = Enabled) {
                    Sw := Off ;
                    format(''motor-stopped \n'')
               } ;
         } ;
   } ;
};
```

Let us take each of the above 3 procedures one by one for abstraction.

- **The motor-unsafe procedure :**

  Figure 6.31 is the procedure for **motor-unsafe()** written using the visual notation. Transforming the Visual "If Then" construct using the "if-1" rule for abstraction[2], we get the transformed specification in Figure 6.32.

[2]Using a rule for abstraction means that we consider the transformation rule for the purpose of abstraction. As we saw in chapter 5, we could use any transformation rule for both refinement and abstraction as long as it is applicable.

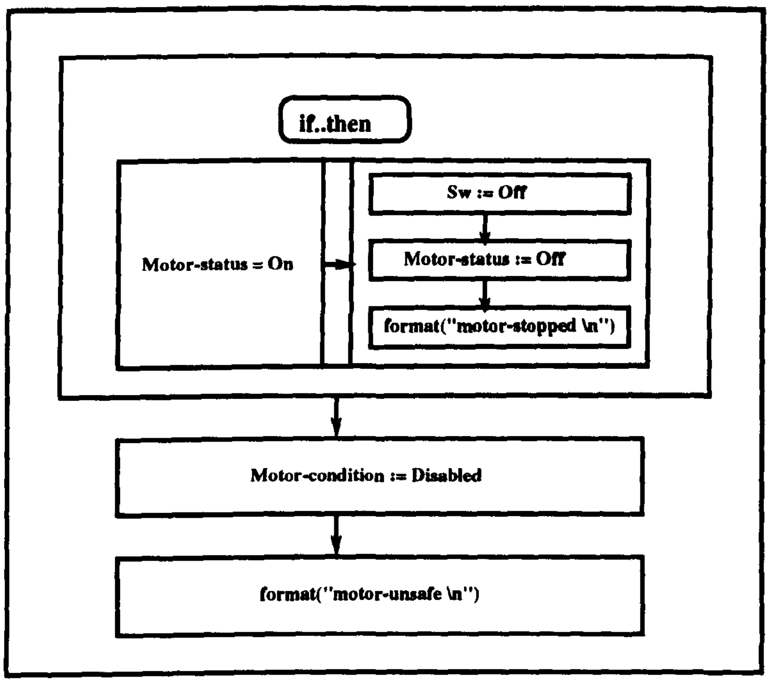

if..then
Motor-status = On
Sw := Off
Motor-status := Off
format("motor-stopped \n")
Motor-condition := Disabled
format("motor-unsafe \n")

Figure 6.31: The Procedure as it is in Tempura

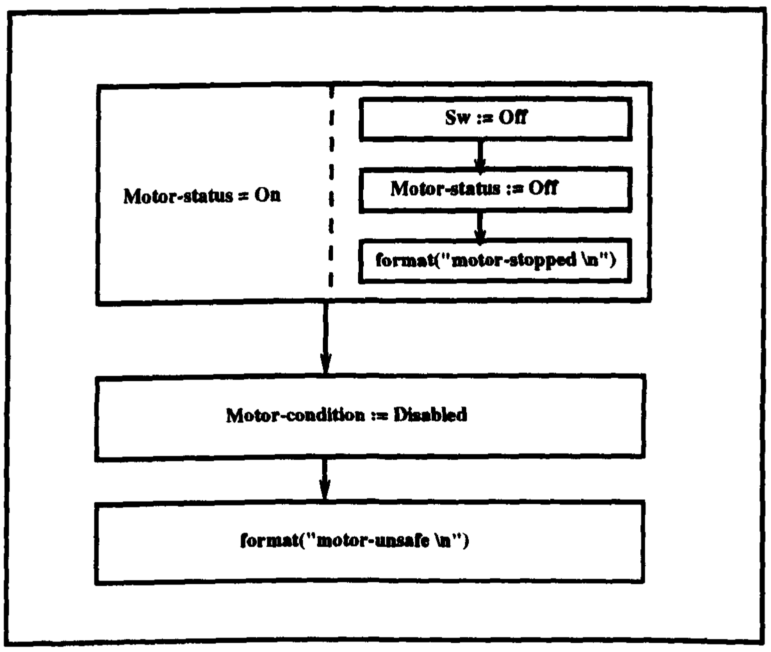

Motor-status = On
Sw := Off
Motor-status := Off
format("motor-stopped \n")
Motor-condition := Disabled
format("motor-unsafe \n")

Figure 6.32: After Visual Abstraction using if-1

Some "chop" operators could be replaced by logic conjunction using VisPA Rule 6 resulting in further logic composition. The rule is reproduced below from chapter 5.

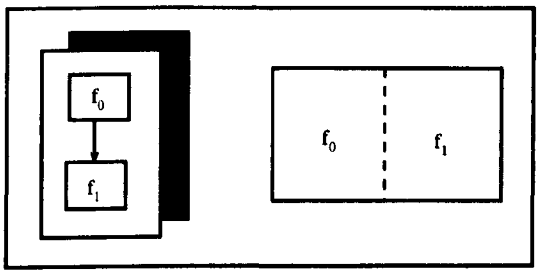


Figure 6.33: Visual Primitive Abstraction Rule 6 : Sequential

Applying this rule, we obtain Figure 6.34.

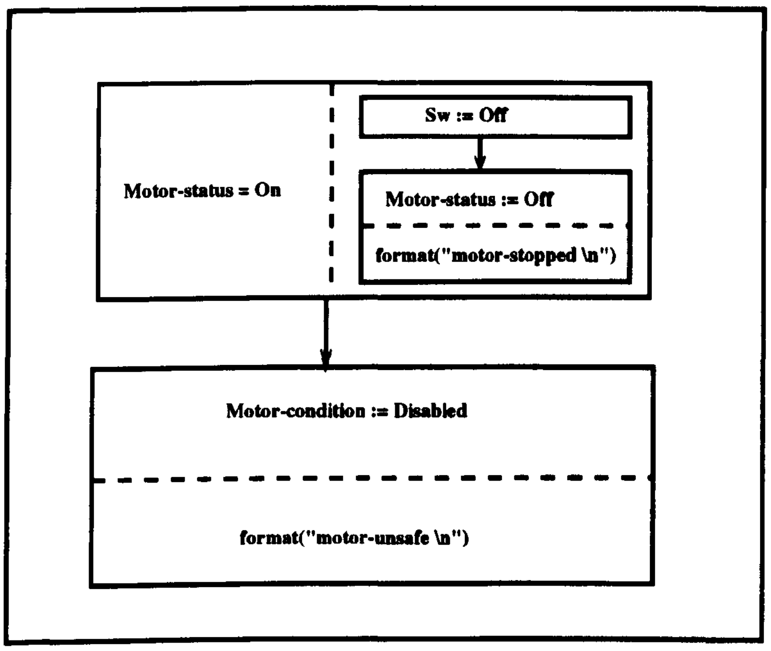

Motor-status = On
Sw := Off
Motor-status := Off
format("motor-stopped \n")
Motor-condition := Disabled
format("motor-unsafe \n")

Figure 6.34: After Logic Composition

There are quite a lot of exception test and handling details in the specification which need to be abstracted away. After doing this, we obtain Figure 6.35. After applying the Visual Primitive Abstraction Rule 1 which extracts a logic formula from an assignment statement, we obtain Figure 6.36.

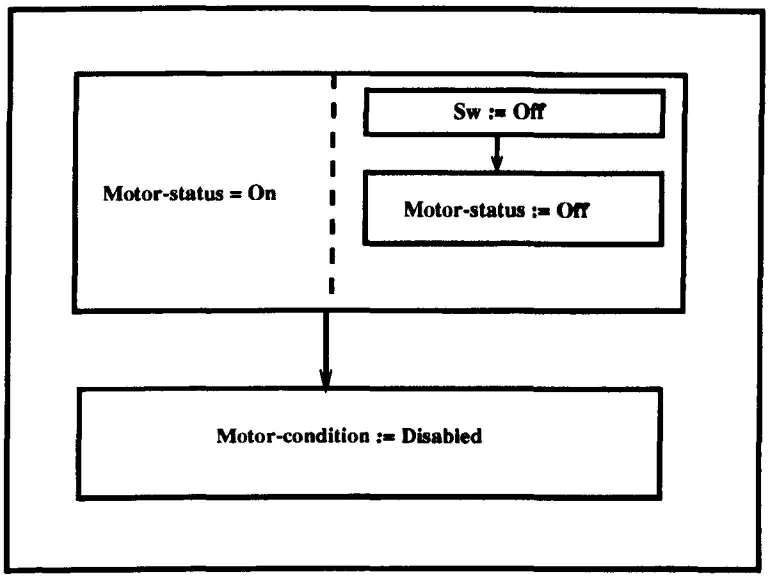


Figure 6.35: After leaving out Unnecessary Details

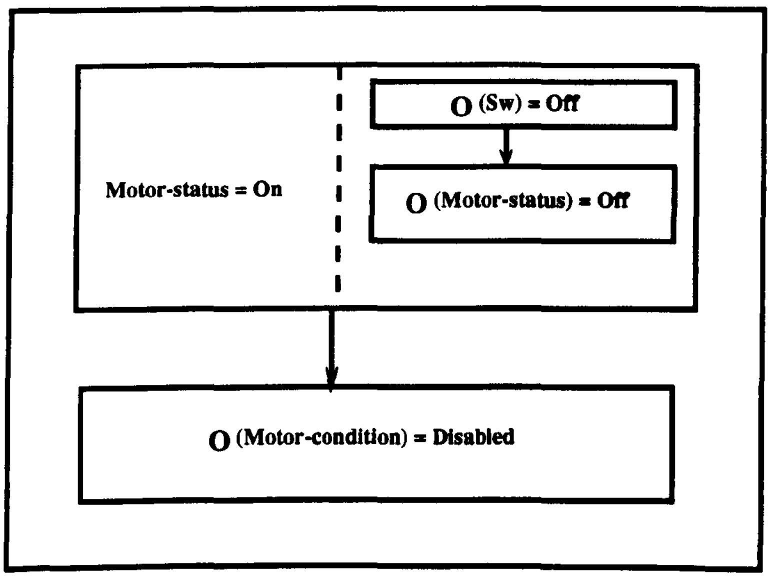


Figure 6.36: After applying the Visual Primitive Abstraction Rule 1

- **motor-safe() procedure**

Figure 6.37 is the procedure for **motor-safe()** written using the visual notation. Transforming the Visual "If Then" construct using the "if-1" rule, we get the

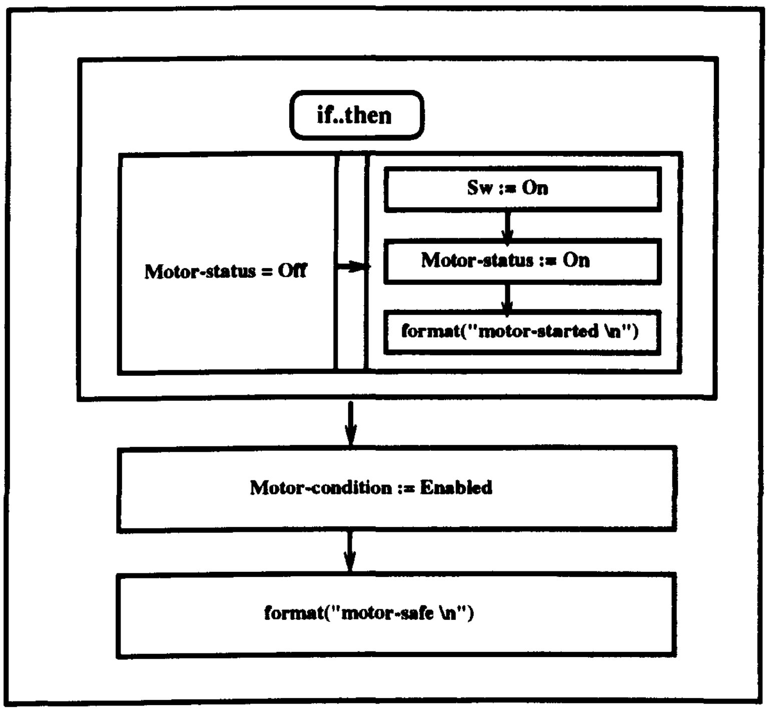


Figure 6.37: The Procedure as it is in Tempura

transformed specification in Figure 6.38. Some "chop" operators could be replaced by logic conjunction using VisPA Rule 6 resulting in further logic composition.

Applying this rule, we obtain Figure 6.39.

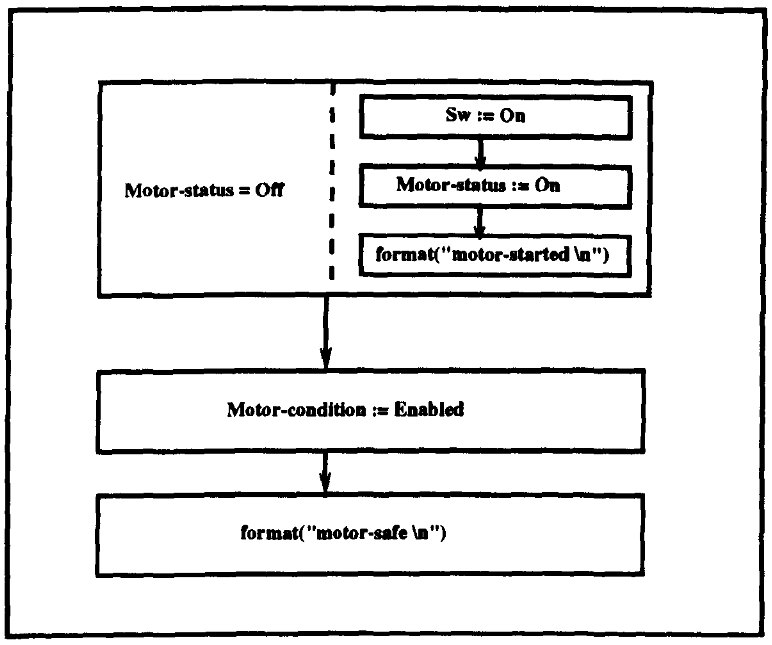


Figure 6.38: After Visual Abstraction using if-1

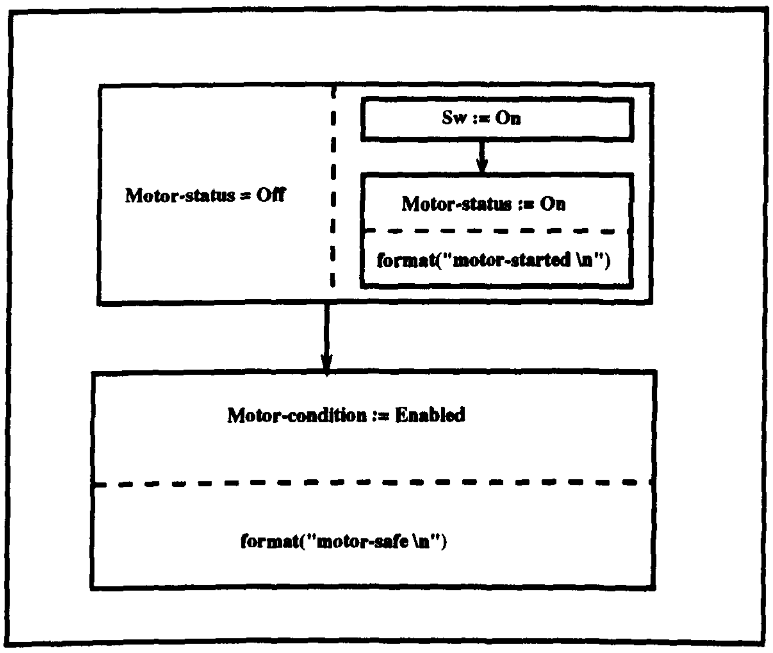


Figure 6.39: After Logic Composition

There are quite a lot of exception test and handling details in the specification which need to be abstracted away. After doing this, we obtain Figure 6.40. After

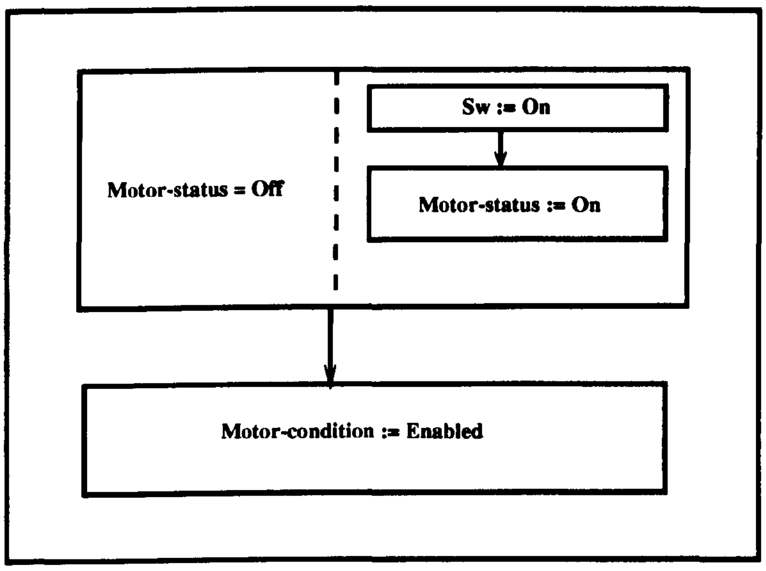


Figure 6.40: After leaving out unnecessary details

applying the Visual Primitive Abstraction Rule 1 which extracts a logic formula from an assignment statement, we obtain Figure 6.41.

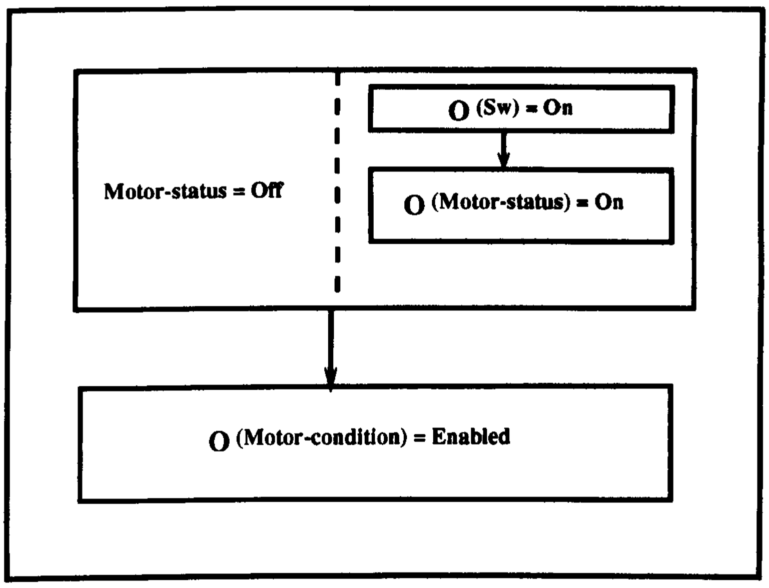


Figure 6.41: After applying the Visual Primitive Abstraction Rule 1

- **set-pump() procedure**

  Figure 6.42 represents the procedure using the visual notation. The zoomable portions in it are given by Figures 6.43 and 6.44.

  The abstractions performed lead to the following specifications, Figures 6.45, 6.47 and 6.48. The final abstracted specification is shown in Figure 6.49. The explanations are provided in the captions associated with the specification figures.

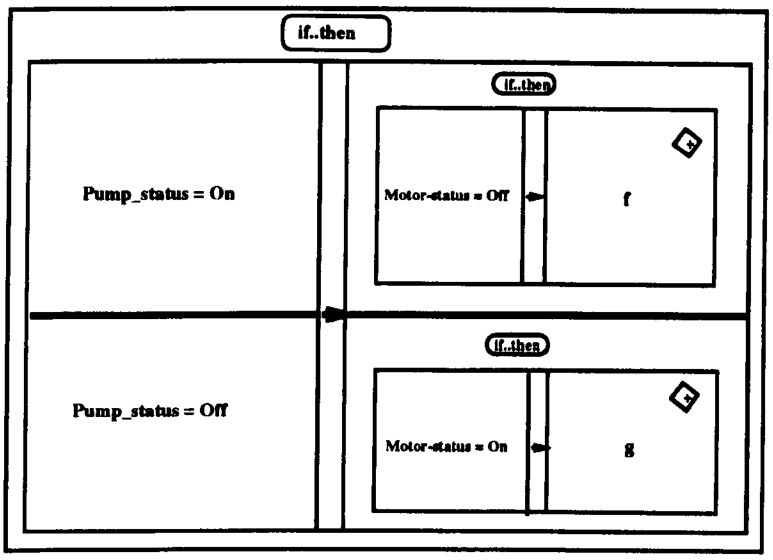


Figure 6.42: The Procedure in Visual Notation

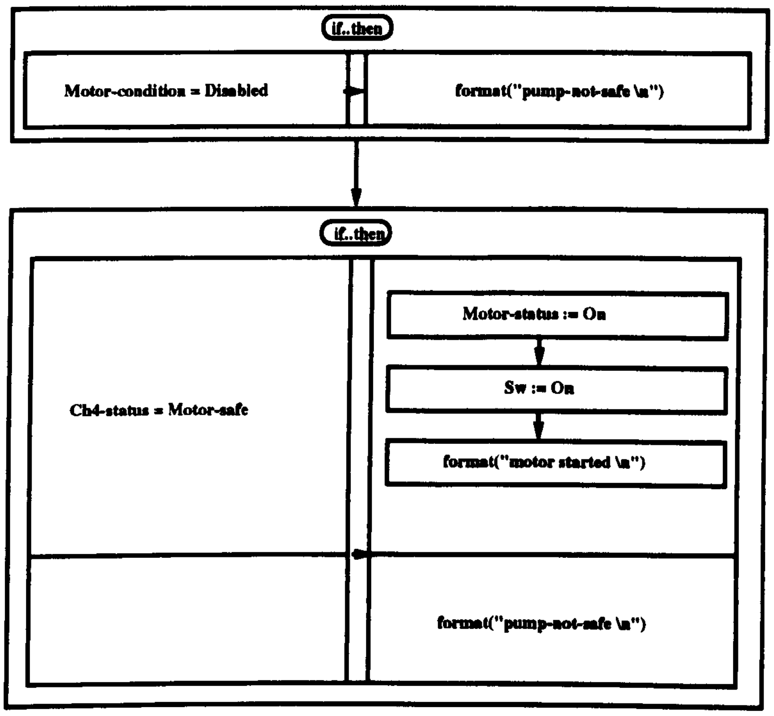

if..then
Motor-condition = Disabled
format("pump-not-safe \n")
if..then
Ch4-status = Motor-safe
Motor-status := On
Sw := On
format("motor started \n")
format("pump-not-safe \n")

Figure 6.43: The Zoomable part 'f'

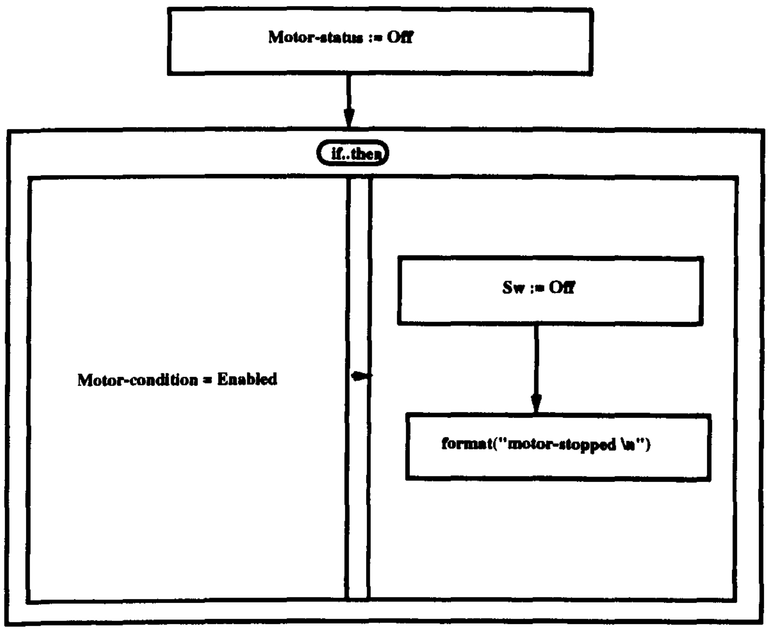


Figure 6.44: The Zoomable part 'g'

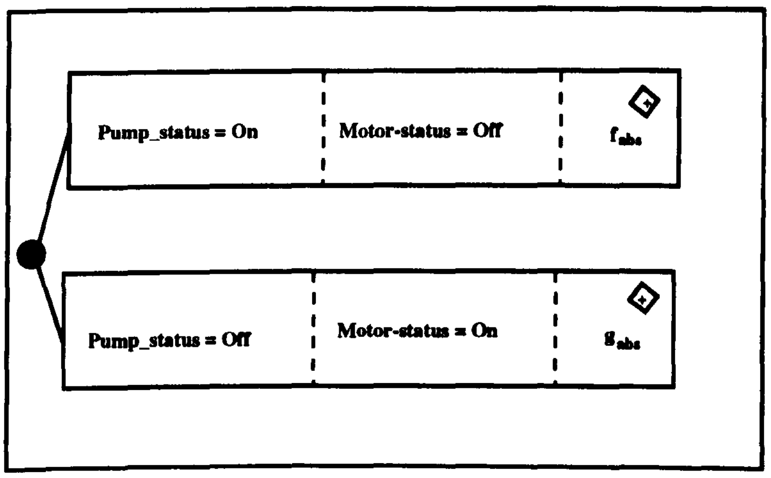


Figure 6.45: The Procedure Abstracted using if-1

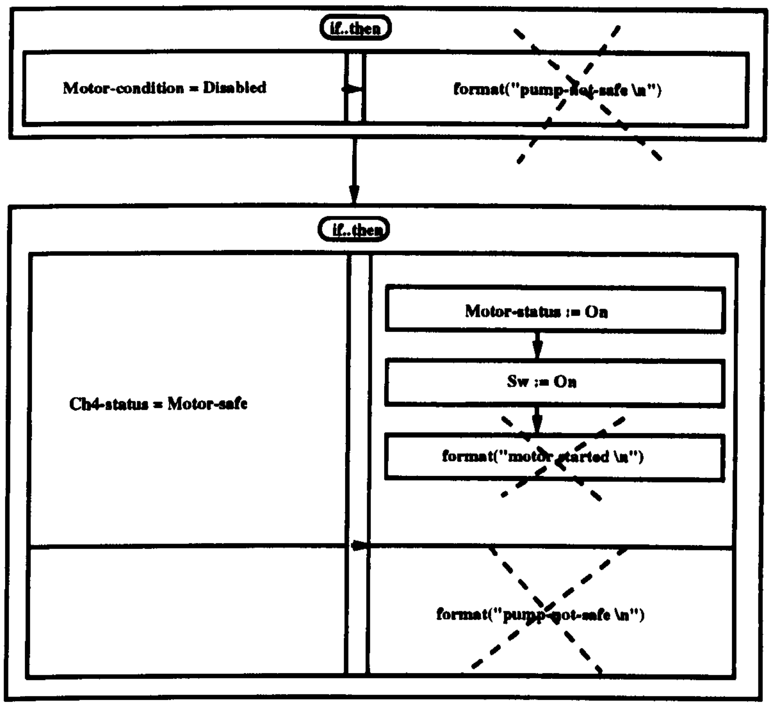

if..then
Motor-condition = Disabled
format("pump-not-safe \n")
if..then
Ch4-status = Motor-safe
Motor-status := On
Sw := On
format("motor started \n")
format("pump-not-safe \n")

Figure 6.46: The Zoomable part 'f' with Unnecessary Details marked with Crosses

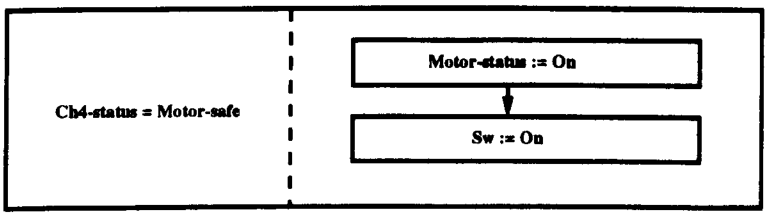


Figure 6.47: The Zoomable part 'f' after Dropping Unnecessary Details and using if-1

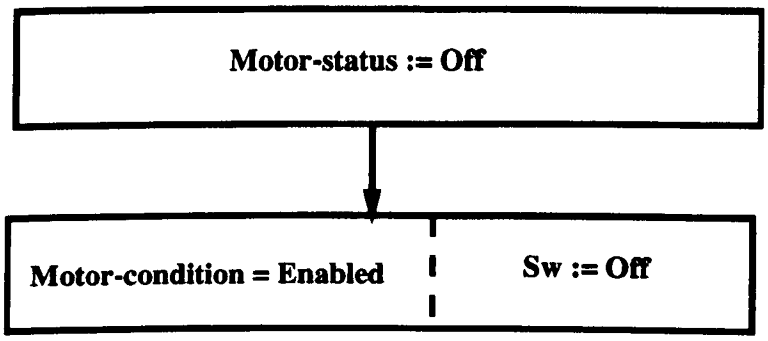


Figure 6.48: The Zoomable part 'g' after Dropping Unnecessary Details and using if-1

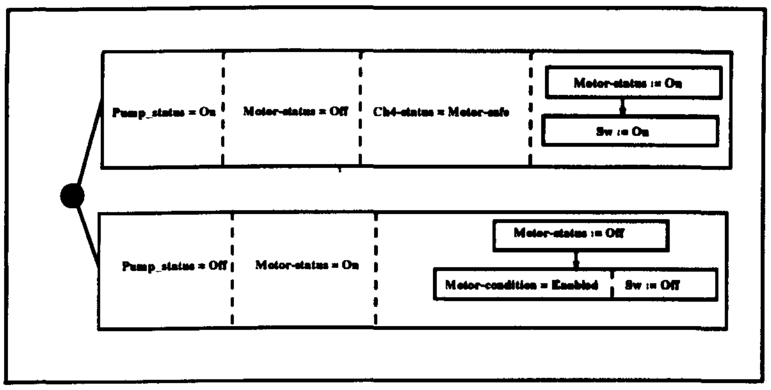


Figure 6.49: The Composed Abstracted Visual Specification for set-pump

We can now abstract assignment statements by applying the Visual Primitive Abstraction Rule 1 on this composed specification.

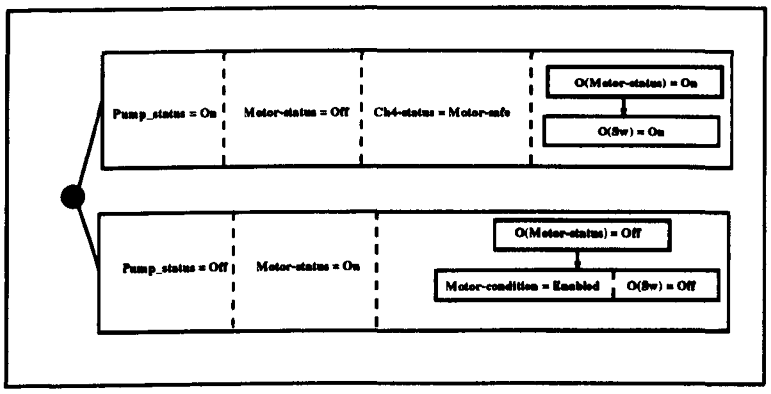


Figure 6.50: The Composed set-pump Specification after Abstracting Assignments

## 6.4 Comments on Visual Abstraction

In this section, we saw how visual abstraction could also be performed in our visual framework to obtain a VisITL specification. The starting point was from a translation of the implementation (i.e., CSL code in our example) to a VisITL form (rather its executable equivalent). Using the visual transformation rules for abstraction, we obtained a VisITL specification. The first step of translating an implementation (in CSL or ADA, C etc.) to Tempura requires some experience in these languages. After this step, the abstraction is possible within our VisITL framework.

We used the zoom feature to manage the specification by abstracting bits of specifications using rules and then composing them. In other words, the approach is scalable to large specifications.

## 6.5 Summary

In this chapter, we saw how both refinement and abstraction in ITL are performed in a visual framework involving VisITL specifications. This framework facilitates a controlled development of systems either in forward or reverse engineering. It was easy to see that the refinements on purely textual ITL specifications are not that easy to keep track of despite good indenting. There were not many visual cues to rely on, despite good indentation of the textual specification. This is a crucial point in dealing with even larger specifications on which we might end up applying numerous transformations. This adds further strain, especially on a user who is not dealing with mathematical specifications on a regular basis. Even formalists would be hard-pressed to keep track of the development process especially when automated tools are being used on ITL specifications. We also saw how visual abstraction could also be performed in the visual framework to obtain a VisITL specification. The starting point was from a translation of the implementation (i.e., CSL code in our example) to a VisITL form (rather its executable equivalent). Using the visual transformation rules for abstraction, we obtained a VisITL specification. The first step of translating an implementation (in CSL or ADA, C etc.) to Tempura requires some experience in these languages. After this step, the abstraction is possible within our VisITL framework. We also saw additional benefits of a visual framework for a formal method like ITL including scalability. In the following chapter, I will give an account of the implementation of the VisITL tool and suggestions for future work.

# Chapter 7

# Implementation

## 7.1 The Visual Tool

This chapter gives details of the VisITL tool implemented during the course of the development of the visual language, VisITL. Also given is an insight into the experience gained and the feedback obtained for the development of the visual language.

### 7.1.1 Block Diagram of the VisITL Tool

The VisITL tool has been implemented in Tcl/Tk. Figure 7.1 gives a block diagram of the tool implementation. The **VisITL** tool has all the standard drawing tool features of the **tkpaint** tool. Apart from this, it also provides special buttons on the tool menu for several VisITL constructs so that they could be drawn on the canvas by drag-and-drop. Once VisITL specifications are constructed, they could be transformed using the **VisITL-Funcs** menu. As depicted in Figure 7.1, these functions include refinement in the visual framework, abstraction in the visual framework[1], the possibility to convert a VisITL specification to its ITL equivalent as well as pretty-printing[2] a VisITL specification. The **VisITL-Help** menu provides help on VisITL constructs, informal semantics of constructs, help on refinement and so on.

[1]not implemented in the current version

[2]In this context, meaning that the diagram is made neat by centering formulas in boxes etc.

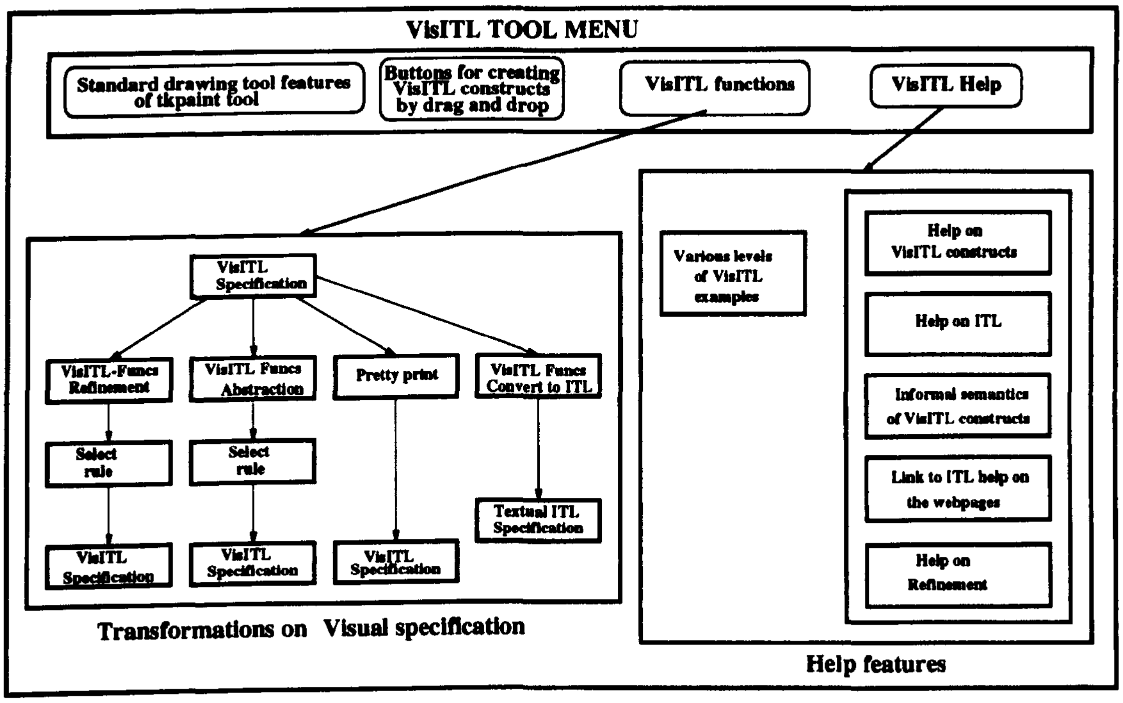

VisITL TOOL MENU
Standard drawing tool features of tkpaint tool
Buttons for creating VisITL constructs by drag and drop
VisITL functions
VisITL Help
VisITL Specification
VisITL-Funcs Refinement
VisITL Funcs Abstraction
Pretty print
VisITL Funcs Convert to ITL
Select rule
Select rule
Textual ITL Specification
VisITL Specification
VisITL Specification
VisITL Specification
Transformations on Visual specification
Various levels of VisITL examples
Help on VisITL constructs
Help on ITL
Informal semantics of VisITL constructs
Link to ITL help on the webpages
Help on Refinement
Help features

Figure 7.1: A Block Diagram of the VisITL Tool

The Figure 7.2 shows a screen dump of the tool. The drawing portion of the tool is an extended version of the **tkpaint** tool [146] freely available. **tkpaint** is a graphics program based on the canvas widget of the tool command language, Tcl/Tk. It runs on several platforms including Linux, Windows 95, NT and Solaris. It is very easy to learn and use. Hence, it was decided to base the drawing feature of the **VisITL** tool on it. Developing another drawing tool itself would not not have been possible due to time constraints apart from the fact that it would have amounted to "re-inventing the wheel". In other words, the **tkpaint** tool was an "off-the-shelf" component for these research purposes. I could therefore start implementing VisITL constructs rather than worry about implementing graphical primitives like lines, rectangles and so on.

### 7.1.2 Features of the VisITL Tool

The following is a description of the features of the tool implemented.

- **Draw visual ITL specifications easily by selecting the visual constructs on the tool bar/menu**

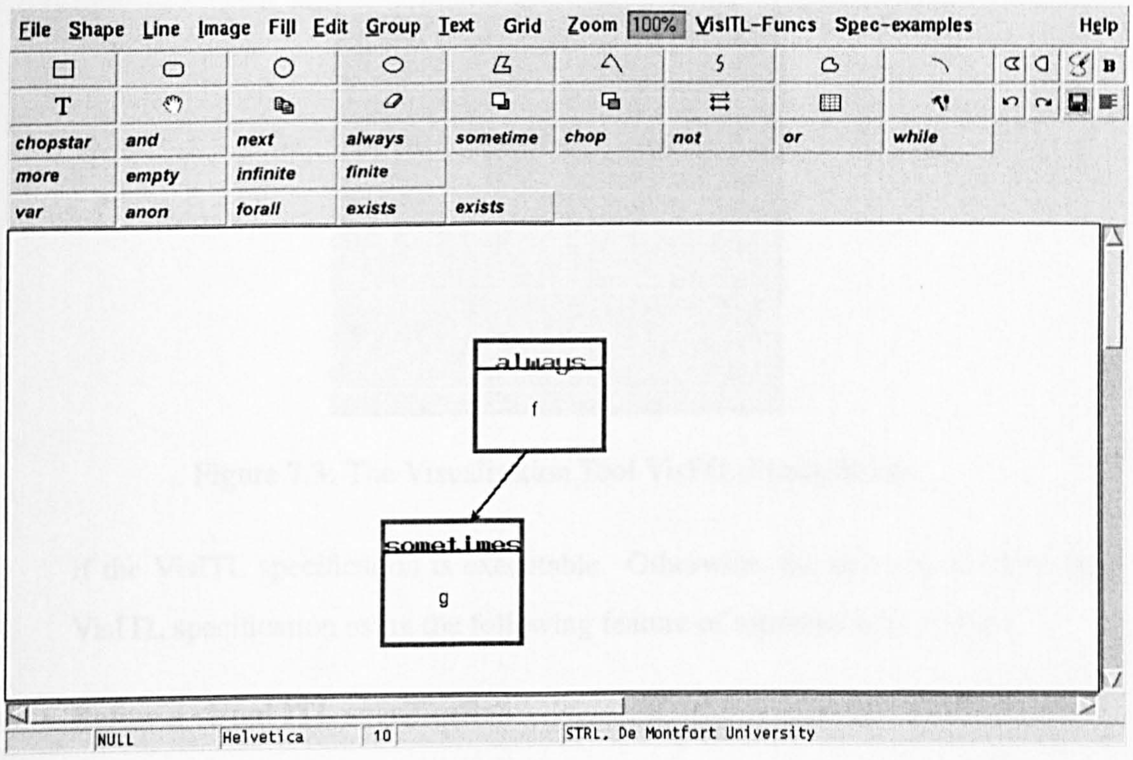

File Shape Line Image Fill Edit Group Text Grid Zoom 100% VisITL-Funcs Spec-examples Help
chopstar and next always sometime chop not or while
more empty infinite finite
var anon forall exists exists
always
f
sometimes
g
NULL Helvetica 10 STRL, De Montfort University

Figure 7.2: The Visualisation Tool for ITL

The buttons shown on the VisITL tool could be clicked and then, using the mouse, the construct could be automatically drawn by first clicking (the left-mouse button) on the canvas to select the left-top corner for the box and then dragging the mouse and releasing it at the position for the right-bottom corner for the box. This way, the size of the box could be adjusted. The buttons for VisITL constructs are provided to speed up the process of constructing a specification.

- **Convert the visual ITL specification to a textual form by the click of a button**

  The VisITL specification created could be converted to its equivalent textual ITL form by choosing an option in the menu button **VisITL-Funcs** on the toolbar. This would enable other tools in the workbench to be integrated in this visual framework. Also, the feature could be used to automatically generate a documentation with textual specifications. The Figure 7.3 shows the options available in the **VisITL-Funcs** menu. The **Convert to** option allows conversion to ITL in the current version. A future option could allow a direct conversion to Tempura

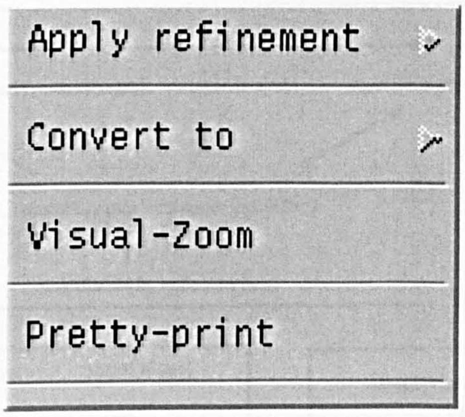


Figure 7.3: The Visualisation Tool VisITL-Funcs Menu

if the VisITL specification is executable. Otherwise, the user has to refine the VisITL specification using the following feature of refinement in VisITL.

- **Refine a visual ITL specification**

  An option in the **VisITL-Funcs** menu allows the user to refine a VisITL specification choosing a refinement rule from a rule repository.

- **Help feature for VisITL tool**

  This provides information on visual ITL constructs together with informal semantics in a visual form. This will aid the user in learning more about the formalism apart from providing help during the development process.

- **Sample VisITL specifications**

  This provides some simple example specifications in VisITL for the user.

### 7.1.3 Some Core Procedures of the VisITL Tool

This section gives a brief insight into some of the key procedures of the VisITL tool. Since some of the procedures are inter-dependent, they are explained with dependency graphs.

- **The drawing procedures**

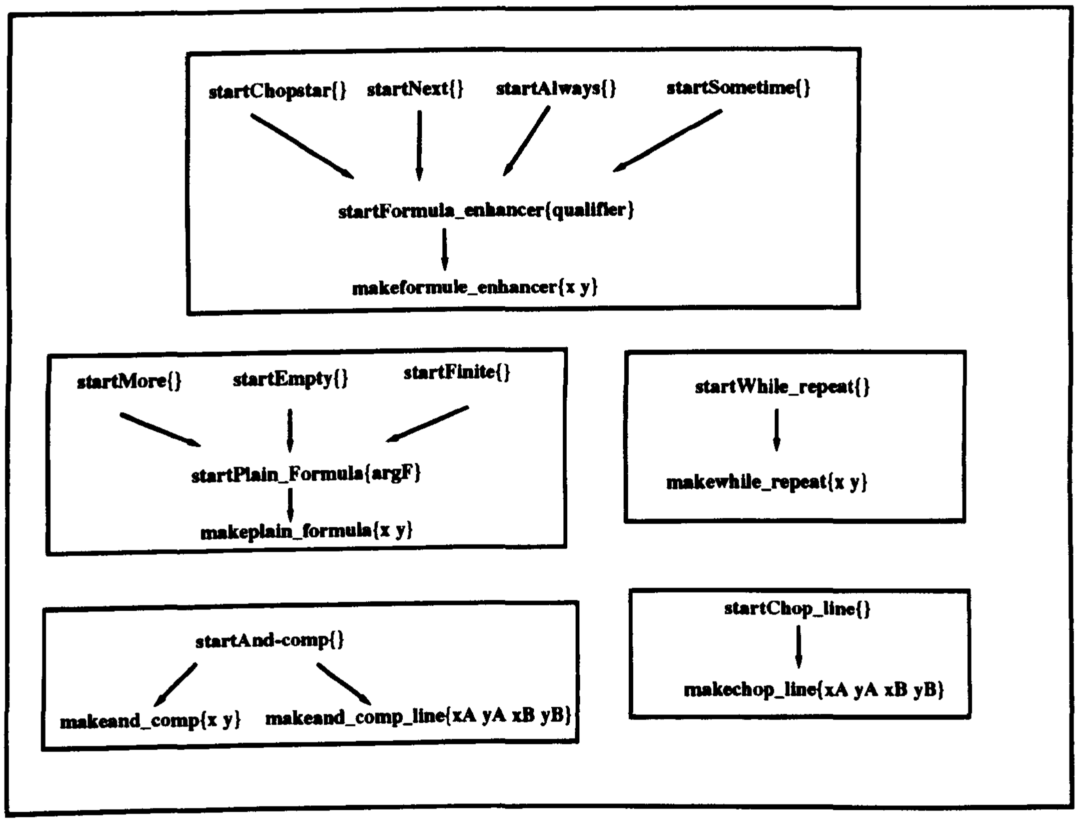


Figure 7.4: The Dependencies of some Drawing Procedures

Figure 7.4 shows a dependency graph of some of the drawing procedures. The procedure proc startFormula_enhancer{qualifier} implements the feature of automatically creating VisITL constructs on the drawing canvas by using the buttons provided on the toolbar. The "qualifier" is the argument which gets supplied automatically by clicking the menu button. For example, clicking on the "Always" button provides "Always" as the qualifier to the procedure to allow it to draw this construct automatically. As the dependency graph shows, the procedures for "Chopstar", "Next", "Always" and "Sometime" depend on proc startFormula_enhancer{qualifier}. This procedure sets parameters for primitive constructs like line, rectangle and text involved in the creation of these visual constructs[3] and also notes the starting x-y co-ordinates selected on the canvas. This

[3]Note : The implementation is not based on using labels but on the earlier visual notation which used a separator line between the formula and the qualifier.

procedure then passes the co-ordinates of the positions at which the mouse was released to the makeformula_enhancer{x y} which completes the drawing of the figure. Depending on the qualifier passed as argument, one could change settings for the fonts or colour for the visual construct. The proc startPlain_formula{argF} implements the drawing of simpler formulas like "empty", "finite" etc. which are just text enclosed in boxes.

- **Procedures for conversion to textual ITL**

  These procedures are required so that the user can produce the textual form of ITL and derive the benefit of existing tools in the ITL-workbench like Tempura, PVS and so on which depend on textual ITL input. This way the user can benefit from all current and any future support tools for textual ITL without knowing ITL syntax but by just using visual ITL in the graphical VisITL tool conveniently.

  Figure 7.5 shows the dependency of procedures implementing this feature. Proc

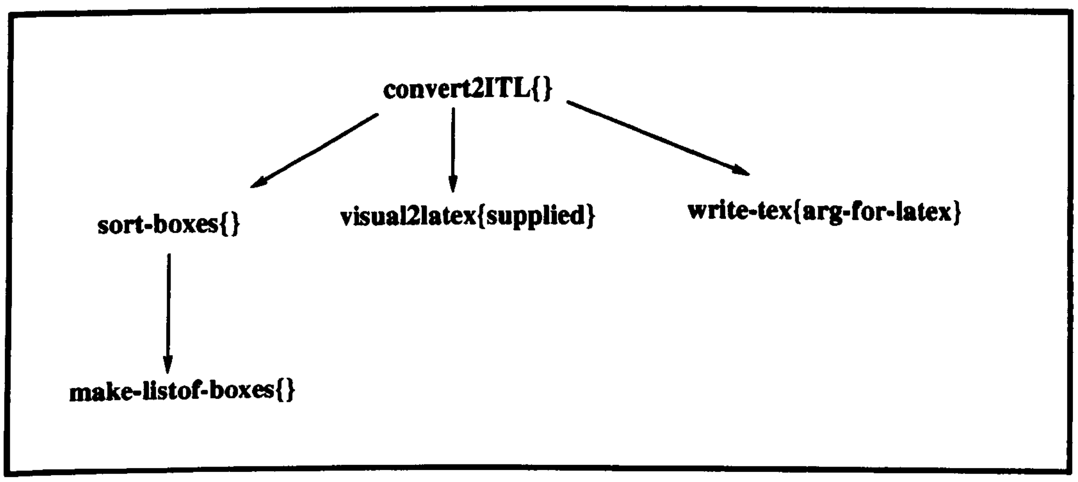


Figure 7.5: The Procedures for Converting to Textual ITL

convert2ITL{} converts the VisITL formula to a equivalent textual ITL form. It depends on procedures proc sort-boxes{}, proc visual2latex{supplied} and proc write-tex{arg_for_latex}. Proc sort-boxes{} ensures that the textual formula generated is correct irrespective of the order in which the boxes were drawn (inner boxes first or outer boxes first or in some mixed order). This was necessary as the implementation of the tkpaint tool keeps information of primitive items in the order they were drawn. Hence, a new list of primitive items had to be generated according to the semantics of the VisITL language. Also since constructs like "chop" could be drawn by the user in any of the ways shown in Figure 7.6 which are all legal in this framework, this generated list had to be appropriately modified. Thus, the semantics of the VisITL language comes into considera-

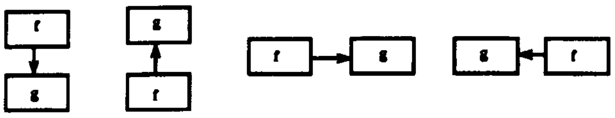


Figure 7.6: Using Chop in Several Forms

tion. Such processing in done in this procedure. The current VisITL tool allows "chop" in all the above forms i.e., with horizontal and vertical arrows, as well as with slanted arrows. The proc visual2latex{supplied} brackets this listing obtained in terms of the number of arguments expected for each visual construct. In other words, this list is transformed into a flattened tree-structure corresponding to the equivalent textual ITL formula. This is passed as arg_for_latex to the procedure proc write-tex{arg-for-latex} which generates a latex program. This program could be used to obtain the textual ITL specification in postscript form. The current implementation automatically generates the postscript and displays it to the user.

- **Procedures for refinement**

Figure 7.7 shows some procedures for refinement and their dependencies. As refinement rules could be visualised as moving sub-formulas around into newer locations within possibly different visual constructs, they depend on procedures like proc sub-ITLformula-in{xA yA xB yB} which was written to find out sub-formulas in given locations, proc visualize_in{xA yA xB yB tree_info} which was written to re-draw the sub-formula in the new location. Other drawing procedures could also be called during refinement as, for example, a chopstar would

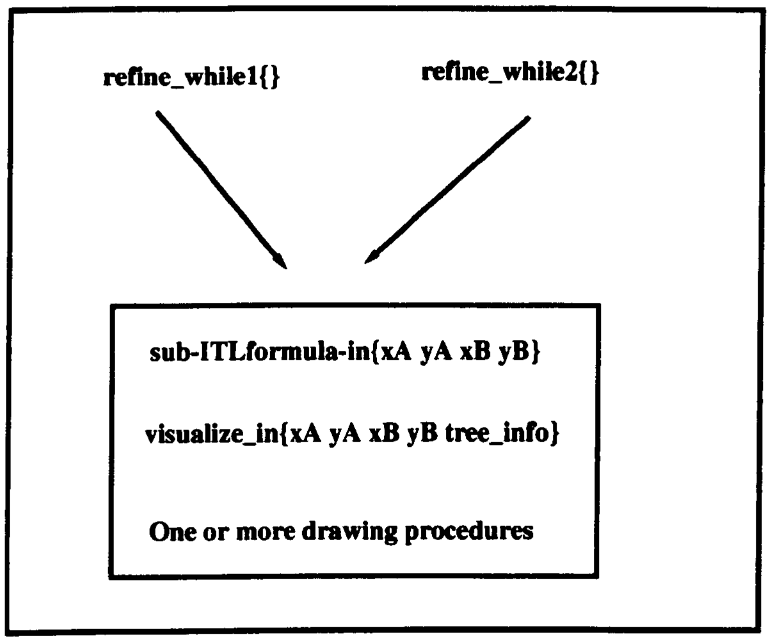

refine_while1{}
refine_while2{}
sub-ITLformula-in{xA yA xB yB}
visualize_in{xA yA xB yB tree_info}
One or more drawing procedures

Figure 7.7: The Dependencies of Some Procedures for Refinement

get refined to a while construct and so on.

Further details on these procedures are given in Appendix B.

## 7.2 Lessons Learnt and Problems Encountered

As already mentioned in chapter 4, expressions were found to be better written as in textual form in order not to introduce too many graphical primitives. Formulae in boxes with a separator line for the qualifiers were sometimes found to be confusing when distinguishing the formula text and the qualifier text. It was decided to incorporate labels into the VisITL language. During implementation, as many constructs as possible were based on a similar format. But, for the sake of intuitiveness, the visual language had to have more appropriate visual constructs for "chopstar", for example. Hence, there was always a compromise that had to be made. Moreover, for some constructs like "while", I chose a simpler form of visual construct so as not to overload the picture with many graphical primitives.

As the visual language for ITL was non-existent at the start of my research, it was difficult to design an implementation. In other words, there was no specification on which to base the implementation. As a result, the aim of my implementation was to investigate and obtain valuable insight into the evolving visual language itself. Therefore, the current version of the implementation uses some older form of visual notations as already mentioned. Modifying the visual constructs implemented to match the current visual language would not be that difficult now but it would have its effect on all other functionality of the tool. Based on the current level of development of the language, better implementations could be designed. One could even go for implementations that abstract away the exact form of the visual notation itself. This focus could not have been entirely implementation as the visual language was evolving. In order to explore many more different implementation algorithms and designs, one could first develop a library of some core functions which could be utilised by all the implementations. This would enable some new implementations to be explored as future work in this area.

As already mentioned in an earlier section, conversion of a VisITL formula to its equivalent textual ITL formula had to consider an algorithm based on all primitive objects on the canvas in terms of the VisITL constructs they constituted and not on the order in which they were drawn. Also, as mentioned earlier, "chop" could be drawn in several ways and hence, the ordering was not purely based on co-ordinates of boxes but sometimes had to be adjusted. The current version of the VisITL tool, as mentioned earlier, allows the drawing of chop in any form i.e., horizontal, vertical or slanted arrow.

## 7.3 Future Work

A textual ITL formula could now be visualised based on the VisITL language. Such a procedure would need an argument which encodes a tree structure for the formula. This procedure would be similar to proc visual2latex{supplied} which constructed a tree structure for the VisITL formula drawn on the canvas. This way, one could incorporate already existing textual ITL specifications into this framework.

Further work could also be done on incorporating restrictions on the attempted Vis-

ITL specifications through appropriate warnings generated by the implementation.

Also, one could create a library of VisITL specifications or specification-chunks which could be dragged and dropped on the canvas to have ready-made blocks of specifications. Then, the general parameters could be mapped to specific identifiers to realize the VisITL specification. This would speed-up the specification process further and hence enable further accessibility of the formal method. One more possibility with implementation, now that no practical distinction between transformation rules for refinement and abstraction are made, is that we could just have one set of transformation rules and use any rule either for refinement or abstraction based on a user selection on a radio-button. In other words, the user could select one rule from a list and apply it on the VisITL specification to either refine or abstract.

With regard to executions of concrete VisITL specifications, further work needs to concentrate on integrating Tempura into the framework to allow the user to visualise simulations graphically.

## 7.4 Chapter Summary

Details of a tool developed for the VisITL language were given. The features of the VisITL tool i.e., drawing, conversion to textual ITL and refinement were explained. Also, with the help feature on the tool, a user could learn VisITL syntax and semantics with examples. Lessons learnt and problems encountered during implementation are explained in order to benefit future work in this area. Conversion of a VisITL formula to its equivalent textual ITL formula had to consider an algorithm based on all primitive objects on the canvas in terms of the VisITL constructs they constituted and not on the order in which they were drawn. Also, as mentioned earlier, "chop" could be drawn in several ways i.e., horizontal, vertical or slanted arrow and hence, the ordering was not purely based on co-ordinates of boxes but sometimes had to be adjusted. Core procedures of the implementation are given in the appendix for the sake of future users.

# Chapter 8

# Conclusions

## 8.1 Summary

Chapter 1 contains the background and motivation for the work and an outline of the thesis.

In chapter 2, process models and strategies in software development were explored and various formal approaches for the development of real-time, safety-critical systems introduced. While formal approaches were found to have benefits in development especially for safety-critical systems, they have many drawbacks to overcome especially for enabling them to be more suitable for widespread use. Mentioning the rationale for the choice of Interval Temporal Logic (ITL) as the formalism for the work, the objective of this thesis is to contribute to the development of ITL as a lean formal method i.e., a formal method that is more accessible.

In chapter 3, we saw how visualisation helps in various domains in fostering increased accessibility of information, languages and technology through various means including better communication and ease of use. We examined how we could apply visualisation aids to an ITL-based formal method. Specifically in order to design a visual language for ITL, a design rationale was developed and key requirements to be satisfied by such a visual language identified. To recapitulate, these requirements are briefly summarised here :

Based on a study of visual representations in various areas, the following were identified as key features that should be taken into account while designing a visual language.

- Simplicity

  This is one of the most important features of a good visual notation. The simpler the diagram, the easier and quicker it is to extract the meaning from it.

- Intuitiveness

  The diagram should draw the attention of the reader quickly to the suggested meaning.

- Unambiguity

  The diagram should not be causing any confusion in interpretation.

- Readability

  The diagram should have text put at suitable locations to enhance readability without overcrowding the diagram.

- Communicativeness

  It should be possible to communicate with other people using the diagram. In other words, it should be possible to suitably abstract the diagram, when necessary, and see the new diagram clearly. It should be possible to navigate the diagram contents with ease.

- Manipulatability

  It should be possible to manipulate the diagram to suitable equivalent forms. The meaning of any manipulation should be clear.

- Composability

  It should be possible to suitably compose two diagrams.

- Extensibility

  The visual notation should support extensions to the basic language with mechanisms/suggestions on how to do so.

- Customisability

  It should be possible to customise the basic notation in some restricted ways for which guidelines should be provided.

- Realisability

  It should be possible to realize the visual language conveniently in a suitable tool.

- Scalability

  It should be possible to deal with huge descriptions nearly as conveniently as smaller ones.

- Expressivity

  It should have enough expressivity in terms of available language constructs so that expressing anything in context can be done directly rather than by roundabout methods.

In chapter 4 , we then explored ITL in detail with example specifications highlighting some of the problems with textual ITL. We examined how we could apply visualisation aids to an ITL-based formal method. The approaches until then concentrated on graphical output for simulation through Tempura and work on automata representations for ITL formulae. An interesting possibility as yet unexplored, was the development of a visual language for ITL. Hence a visual language for ITL was developed adhering to the requirements of a good visual language identified earlier. The visual notation is summarised in Figure 8.1 and Figure 8.2 below.

The salient features of the notation can be summarised as follows :

- Expressions are written as in textual ITL.

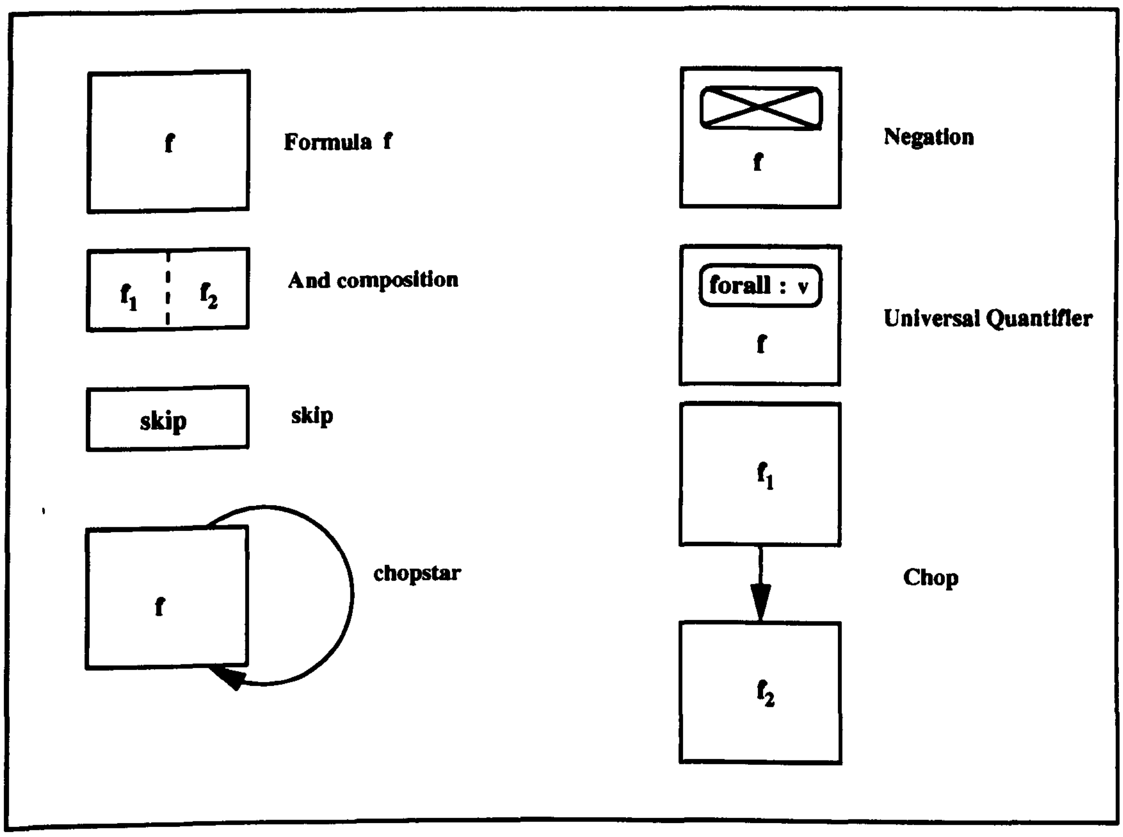


Figure 8.1: Summary of the Visual Notation for Primitive ITL Formulae

- A Formula is enclosed in a rectangular box which also contains a rounded rectangular box for holding any operator information ; the rounded rectangular box maybe omitted if there is no operator involved.
- The "And composition" has dotted lines between the components in the formula.
- The negation has a cross mark in the label portion of the rectangular box.
- The existential quantifier also follows the labelled rectangular box format.
- The chop has a directed arrow between the two rectangular boxes containing the formulae for the left and right subintervals respectively.
- The chopstar has a directed arrow from the rectangular box to itself indicating a looping construct.

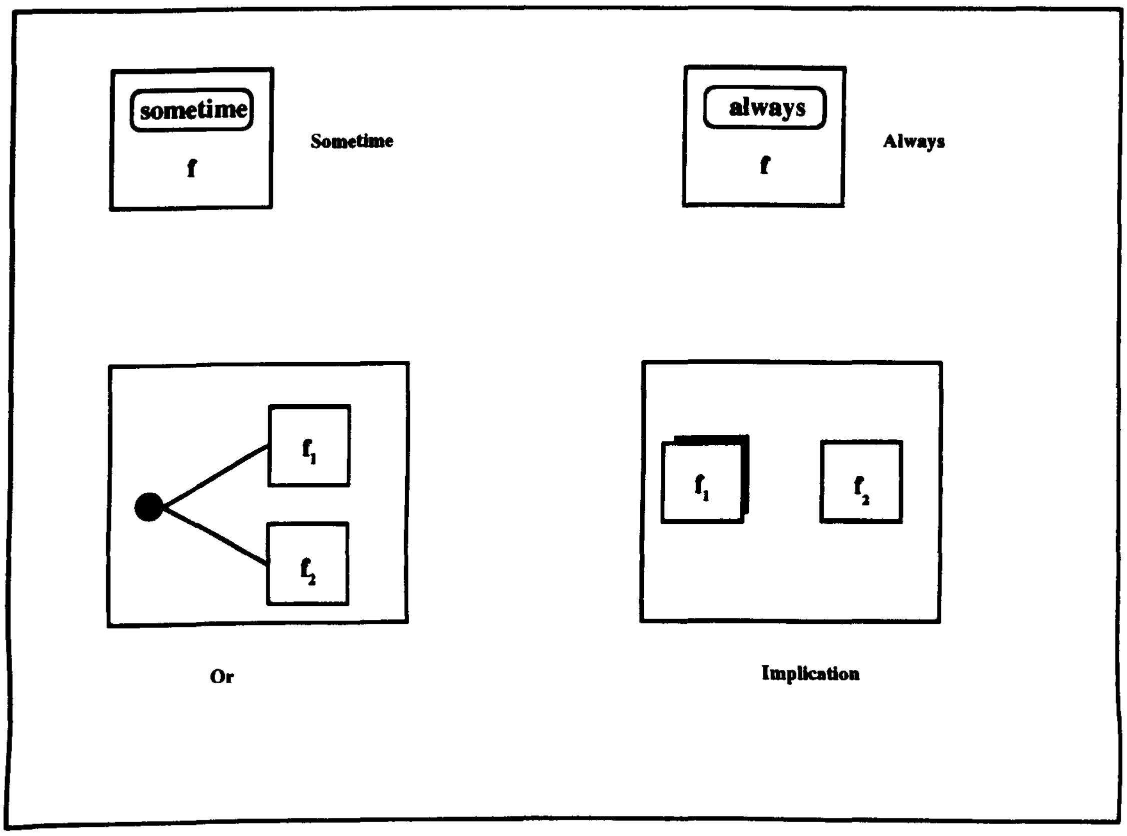


Figure 8.2: Summary of the Visual Notation for Some Abbreviations

The choice of the visual notation has been influenced by various factors as seen in section 4.6. These influences were a direct result of the the important design features identified for visual representations from sectionSect:KeyVis. The following paragraph summarises some key points.

In section 3.1.11, we saw how an Interval Logic, namely GIL [41], visually depicted formulae in the logic. It is dependent on constructing different sub-intervals and then depicting the predicates that are true in such subintervals. Hence, it needed different kinds of graphical primitives like a solid line to represent strong intervals, i.e. non-empty intervals, a double solid line to depict a weak interval, a single arrowhead to indicate a weak search while an interval is constructed, a double arrowhead to indicate a strong search and so on apart from using temporal operator symbols in the visual representation. Also, the placement of the predicate relative to the line depicting the interval has a meaning; for example, if it is the middle of the interval, then, it is an invariant as in Figure 3.7(b). In the context of VisITL language, a much simpler, more intuitive and readable language was needed which also integrated TAM for concrete communication constructs.

It was also considered in section 4.6.3 how predicates were depicted using set theory concepts in a visual logic in [130]. This logic, being based on set theory, has graphical primitives like overlapping boxes which are unnecessary in VisITL, and circles for variables and constants additionally which we found unnecessary as it over-crowds the visual representation as well as unnecessarily complicating the implementation. Moreover, this visual logic is not a temporal logic.

We also saw in chapter 3 how parallel states were depicted in statecharts-based formalisms. The same "dotted-line"representation was chosen for the "and-composition" in VisITL so that no new notation is introduced for similar concepts in other languages. This would help avoid any confusion for users previously accustomed to other languages.

For the ITL "chop", a line with arrow was chosen to represent the meaning of chop i.e., sequential composition. For "chopstar", a loop was depicted around the box. The label in the box was used for readability. The concrete constructs like "While", "If Then" etc. also followed a similar format. This is true for the TAM communication constructs as well. Hence, VisITl was not only made simple, intuitive, readable and unambiguous but also one that integrated abstract constructs and concrete constructs in similar notations. This aids communicativeness between users at all abstraction levels.

The geometrical guidelines for constructing VisITL specifications show how a specification can be manipulated and composed. The VisITL abbreviations allow us to manipulate diagrams to suitable equivalent forms. In a similar way, the user can add new abbreviations as a way of extending the syntax and thus customise the notation. Chapter 6 demonstrates the scalability of this approach. Chapter 7 demonstrates the realisability of this approach. The VisITL language derives its expressivity from ITL.

ITL examples given earlier were re-worked using the VisITL language constructs. Although with VisITL, we had a graphical notation for ITL, we still lacked a visual framework in which formal development could take place in an accessible manner.

In chapter 5, in order to facilitate the development process using the VisITL language, visual refinement and abstraction rules were introduced with examples. Thus a visual framework for both forward and reverse engineering in a formal visual framework supporting both convenience and accessibility are developed. The notation for visual refinement and abstraction is summarised below :

Figure 8.3 denotes a transformation (which can be either refinement or abstraction) visually. It suggests intuitively that $f_2$ contains more details than $f_1$ because of the shaded box. The rule could be applied either during forward or reverse engineering.

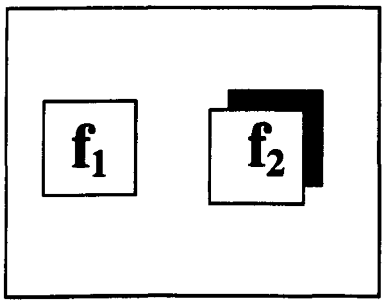


Figure 8.3: Visual Refinement

In chapter 6, case studies showing how the visual framework supported both refinement and abstraction were developed. Also an additional possibility with respect to the visual notation while developing the case studies were explored, like for example, resolving non-determinism through geometrical considerations. Currently, in the visual framework, it was decided to use design decision rules to resolve non-determinism rather than attach meaning to geometrical features in VisITL. This lets the user be in control rather than automatically resolving conflicts which the user may fail to notice.

In chapter 7, the experimental VisITL tool that was built and used in the process was described in detail giving directions for further implementation efforts.

To summarise, we state that a simple, intuitive and readable visual language has been designed which enjoys the benefits of being a formal language. The supporting visual framework of refinement and abstraction enables easy manipulatability as well as communicativeness. The visual notations for abstract VisITL specifications as well as the concrete ITL and TAM constructs are integrated in a similar format. The VisITL tool demonstrates realisability. Extensibility, customisability, composability and scalability are demonstrated through examples and case-studies.

## 8.2 Review of Work Done and Comparisons to Related Work

This work aimed to contribute to the development of ITL as formal method for the development of critical systems. Towards that goal, the research explored visualisation approaches for increasing the accessibility of the formal method. The visual framework that has evolved is an exciting way forward in this direction. A valuable foundation has been laid by designing a visual language as well as evolving a framework in which development can be carried out. This approach insulates the user from as many formal notations and difficulties as possible in a user-friendly manner. Valuable feedback obtained from the tool that was developed for experimentation is passed on to guide further explorations for implementation and development of the visual language itself.

Related work done by others include the Visual Logic of [130], Graphical Interval Logic (GIL) [41] and statecharts-based formalisms such as those supported by Statemate and Stateflow.

The Visual Logic of [130] aimed to make formal specifications accessible to non-programmers by using familiar notions of set theory, such as set inclusions and their graphical representations. They have made attempts to link their visual logic specifications to executable specifications in Prolog. VisITL is a similar effort based on ITL though not following the concepts of set theory. VisITL uses simpler intuitive visual notations suitable to ITL syntax and semantics. In VisITL, the number of graphical primitives is deliberately optimised keeping implementation issues in mind. Therefore, it is a much more practical approach and more suitable to ITL syntax.

The Graphical Interval Logic, on the other hand, is intuitive but uses a different approach, one that depicts intervals as horizontal lines. This approach is unsuitable to VisITL as one of the aims was to seamlessly merge VisITL with its executable subset i.e., Tempura. Further, integration with the Temporal Agent Model (TAM), allowed us to incorporate concrete communication constructs in the same framework. With both forward and reverse engineering supported by the visual refinement and abstraction rules developed in this work, a unified visual framework has now resulted in which formal development of systems is possible in a intuitive and accessible manner as demonstrated through examples and case-studies.

Statecharts-based formalisms, like those supported by Statemate and Stateflow, are supported with verification capabilities through model-checkers. This model-checking capability is lacking today in the VisITL framework. When this capability is realized, this framework will have the benefit of the model specified in VisITL checked against a property also specified in the same language. This is unlike in statecharts-based formalisms where the properties are specified in a temporal-logic based language while the model is statecharts-based. Also, in stateflow, non-determinism is resolved through geometrical considerations in the stateflow diagram whereas no such automatic resolution of non-determinism is provided in the VisITL framework. This is deliberately done so in VisITL to avoid the user getting confused by any unintended semantics being automatically given to the resulting implementation. The user is given the choice of applying his own design decision rules instead.

The VisITL framework is therefore not just a visual language for ITL, but is supported by a development process encompassing both forward and reverse engineering through visual refinement and abstraction rules. The user need not know the notations of ITL but can use the more intuitive VisITL language in the convenient VisITL tool and derive the benefit of all the support tools in the ITL-workbench. Therefore, the VisITL specifications can be validated using the ITL proof system. The executable specifications obtained by using refinement rules, could be explored by simulation using Tempura. Hence the VisITL framework which has been developed is a powerful way of formally and visually capturing requirements, validating them, exploring them by simulation as well forward or reverse engineering them in an accessible way.

## 8.3 A Final Note on the Visual Notation and the Visual Framework

The visual notations for Negation, Chop and Chopstar use simple intuitive concepts to depict the meaning of the formula. The "And-composition" uses dotted lines similar to statecharts-based formalisms. The abstract and concrete constructs both follow the box-format. A label within the box aids readability. The label could also hold icons as in the case of TAM constructs to be more visual. All these constructs adhere to the design principles identified in section 3.2. For refinement in the visual framework, we depicted the refinement of a formula intuitively using a shaded box. Through examples and case studies, we saw how the visual framework for formal systems development using ITL can be done in an accessible manner. The implementation of the VisITL tool demonstrated realisability.

## 8.4 Further Work

Refinement strategies could be built-into the VisITL tool to aid the user in performing refinement. Work regarding more advanced features like integration with a proof tool like PVS for constructing proofs could also be carried out. Also, in order to perform verification tasks on VisITL specifications, a model checker for ITL is necessary for integration into the VisITL framework. Work in this area is currently underway [117, 118]. A model checker would provide counter-examples if the property is found to be violated by the model in which case, the user could be shown the scenario leading to property violation through animation in Tempura. Further development with regard to implementation should be carried out with more case-studies supported by industry. This will test the visual language and the framework which has been developed with real users. An investigation should be carried out with a carefully prepared questionnaire to obtain feedback from such users to enable further development and appropriate customisation of the VisITL tool. This would help us build on the initial encouraging results for its evaluation. This would also aid, in general, further development of the formalism and associated tools in the workbench in a direction towards more accessibility.

## 8.5 Conclusions

It is hoped that this thesis has made a useful contribution to enable ITL to be used by a wider spectrum of users. It is also hoped that further developments in the area of ITL, and particularly VisITL, will see a growing number and a wider spectrum of users.

# References

1. Abadi, M. and Manna, Z., "Temporal logic programming", Journal of Symbolic Computation, 8(3), pages 277-295.
2. Agresti, W.W., "New paradigms in software development", IEEE Computer Society Press, 1986.
3. Alur, R., Courcobetis, C. and Dill, D.L., "Model checking in dense real time", Information and Computation, 104(1):2-34, 1993.
4. Anscombe, F.J., "Graphs in Statistical Analysis", American Statistician 27, pp.17-21, February (1973).
5. Ashworth, C.M., "Structured Systems Analysis and Design Method (SSADM)", Information and Software Technology, 1983, 30(3), 153-63.
6. Auermeier, B., Kemmerer, R., "RT-ASLAN : a specification language for real-time systems", IEEE Trans. on Software Eng., 12(9):879-889, September 1986.
7. Back, R.J.R., "A calculus of refinements for program derivations", Acta Informatica, 25:593-624, 1988.
8. Back, R.J.R., "Refinement calculus, part II : Parallel and reactive programs", In de Baker, de Rover, and Rozenberg, editors, Stepwise Refinement of Distributed Systems, volume 430 of LNCS, Springer Verlag, 1989.
9. Back, R.J.R. and Wright, J., "Refinement concepts formalised in HOL", Formal Aspects of Computing, 3(4), 1990.

10. Baudinet, M., "Logic Programming Semantics : Techniques and Applications", Ph.D. Thesis, Computer Science Dept., Stanford University, Stanford, C.A..

11. Beek, M. von der, "A comparison of stateflow variants", In L.de Roever and J. Vytopil, editors, Formal techniques in real-time and fault tolerant systems", volume 863 of LNCS, pages 128-148, Springer-Verlag, 1994.

12. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., And Yi, Wang, "Uppaal - a Tool suite for Automatic Verification of Real-Time Systems", In Hybrid Systems III, volume 1066 of LNCS, pages 232-243, Springer Verlag, 1996.

13. Bergstra, J. and Klop, J., "Algebra of communicating processes with abstraction", Journal of Theoretical Computer Science, 37:77-121, 1985.

14. Berry, G., "Another look at real-time programming", Proceedings of the IEEE, 1991, 79.

15. Berthomieu, B., and Diaz, M., "Modelling and Verification of Time Dependent Systems using Timed Petri Nets", IEEE Transactions on Software Engineering, 17(3):259-273, March 1991.

16. Björner, D. and Jones, C.M, "The Vienna Development Method : The Meta-Language", published by Springer-Verlag as part of the series Lecture Notes in Computer Science, Vol.61, 1978.

17. Boehm, B., "A spiral model of software development and enhancement", IEEE Computer, May 1988, pages 61-72.

18. Bolognesi, T., Brinksma, E., "Introduction to the ISO specification language LOTOS", Computer Networks ISDN systems, 14(1987), pages 25-59.

19. Bonewitz, Ronald L., "TEACH YOURSELF - The complete guide to understanding and writing hieroglyphics", First Edition, 2001, Hodder and Stoughton Ltd., UK.

20. Bowen, Jonathan P. and Hinchey, Michael G., "Seven More Myths of Formal Methods", IEEE Software, Vol.12, No.4, July 1995.

21. Boyer, R. and Moore, J., "A Computational Logic Handbook", Academic Press, New York, 1988.

22. Bremond-Gregoire, P., Choi, J.Y. and Lee, I., "The soundness and completeness of ACSR", Technical Report MS-CIS-93-59, Univ. of Pennsylvania, June 1993.

23. Budgen, David "Software Design", Addison Wesley publication company, 1995.

24. Burns, A., and Wellings, A., "HRT-HOOD : A Structured Design Method for Hard Real-Time Systems", Elsevier, 1995.

25. Büssow, Robert, Gieser, Robert and Klar, Marcus, "Specifying Safety-Critical Embedded Systems with Statecharts and Z : A Case Study", in Egidiano Astesiano, editor, Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering - FASE '98, vol.1382 of LNCS, pages 71-87, Springer-Verlag, 1998.

26. Cau, A., Czarnecki, C., Zedan, H., "Designing a Provably Correct Robot Control System using a 'Lean' Formal Method", In the proceedings of the 5th International Symposium, FTRTFT'98, Lyngby, Denmark, September 1998, volume 1486 of Lecture Notes in Computer Science, pages 123-132.

27. Cau, A., Zedan, H., Coleman, N., Moszkowski, B., "Using ITL and Tempura for large-scale specification and simulation", In proceedings of Fourth Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, pages 493-500, Braga, Portugal, 1996.

28. Cau, A., Moszkowski, Ben, "Interval Temporal Logic Proof Checker", Technical Report available from "http://www.cse.dmu.ac.uk/STRL".

29. Cau, Antonio, and Zedan, Hussein, "Refining Interval Temporal Logic Specifications", In proceedings of the fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software (ARTS'97), LNCS 1231, p. 79-94, Mallorca, Spain, May 21-23, 1997.

30. Chakrapani Rao, Arun, Cau, Antonio and Zedan, Hussein, "Visualization of Interval Temporal Logic", Proceedings of the Fifth International Conference on Computer Science and Informatics, New Jersey, USA, Feb-Mar, 2000.

31. Chang, S.K. “Visual Languages : A Tutorial and Survey”, IEEE Software, 4(1):pages 29-39, January 1987.

32. Chen, Z., “Formal methods for object-oriented paradigm applied to the engineering of real-time systems : A review”, Research Monograph 5, School of Computing Sciences, De Montfort University, Leicester, 1997.

33. Clark, Tony and Evans, Andy, “Foundations of the Unified Modeling Language”, In Proceedings of the 2nd BCS-FACS Northern Formal Methods Workshop, Springer, 1998.

34. Clarke, E.M., and Emerson, E.A., “Synthesis and synchronization skeletons for branching time temporal logic”, In D. Kozen, editor, Logic of Programs Workshop, number 131 in LNCS, Springer Verlag, 1981.

35. Coen-Porisini, A., Kemmerer, R., and Mandrioli, D., “A formal framework for ASTRAL intralevel proof obligations”, IEEE Trans. on Software Eng., 20(8):548-561, August 1994.

36. Cooling, J.E., “Real-Time Software Systems’ : An Introduction to Structured and Object-Oriented Design”, International Thomson Computer Press, January 1997.

37. Damm, W., Josko, B. and Schlör, R., “Specification and verification of VHDL-based system-level hardware designs”, In E.Börger, editor, Specification and Validation Methods, Oxford University Press, 1995.

38. Daws, C., Olivero, A., Tripakis, S. and Yovine, S., “The tool Kronos”, In Hybrid Systems III, volume 1066 of LNCS, pages 208-219, Springer-Verlag, 1996.

39. Day, Nancy, “A Model Checker for Statecharts”, Technical Report No. TR-93-35, Dept. of Computer Science, University of British Columbia, 1993.

40. Dierks, Henning, “PLC-Automata : A New Class of Implementable Real-Time Automata”, In ARTS’97, LNCS, Springer Verlag, May 1997.

41. Dillon, L.K., Kutty, G., Moser, L.E., Melliar-Smith, P.M., and Ramakrishna, Y.S., “Graphical specifications for concurrent software systems”, Proc. of the 14th International Conf. Software Engineering, Melbourne, Australia, pp.214-224, May 1992.

42. Douglass, Bruce Powell, "UML Statecharts", A White Paper in Embedded Systems Programming, January 1999.

43. Downs, E., Clare, P., and Coe, I., "Structured Systems Analysis and Design Method", Prentice-Hall publishers, 215 papges, 1988.

44. Emerson, E.A., "Temporal and modal logic", Handbook of Theoretical Computer Science, van Leeuwen, J.(editor), Noth-Holland, Amsterdam, pages 995-1072.

45. ESA, "HOORA : Hierarchial Object-Oriented Requirement Analysis", 1993, ESA report E2S/OORA/WP1/METH.

46. Esterel webpage at http://www.inria.fr/meije/esterel/.

47. Feyerabend, Konrad and Josko, Bernhard, "A visual formalism for real time requirement specification. In Transformation-Based Reactive System Development", number 1231 in LNCS, pages 156-168, Springer Verlag, 1997.

48. Feyerabend, Konrad and Schlör, "Hardware synthesis from requirement specifications", In Proceedings EURO-DAC with EURO-VHDL 96, IEEE Computer Society Press, 1996.

49. Fisher, M., Kono, S. and Orgun, M.A., Editors, "Editorial : Executable Temporal Logics", J.Symbolic Computation(1996), Vol.22, pages 721-735.

50. Floyd, R.W., "Assigning meanings to programs" in Mathematical Aspects of Computer Science(editor Schwartz, J.T.), Proceedings of Symposia in Applied Mathematics 19, (American Mathematical Society), Providence, Rhode Island, U.S.A., pages 19-32, 1967.

51. Fokkink, Wan and Hollingshead, Paul, "Verification of Interlockings : from Control Tables to Ladder Logic Diagrams", CSR 15-98, Computer Science Report Series, 1998, University of Wales, Swansea, UK.

52. Ford, Lindsey, "How programmers visualize programs", In processings of the Fifth Workshop on Empirical Studies of Programmers, 1993.

53. Formal Methods WWW virtual library at "http://www.afm.sbu.ac.uk/fm".

54. Friel G. and Budgen, D. "Design transformation and abstract design prototyping", Information and Software Technology, 31(9), 707-19.

55. Fuchs, Norbert E., "Specifications are (preferably) executable, Software Engineering Journal, September 1992, pages 323-334.

56. Fujita, M., Kono, S., Tanaka, H. and Moto-oka, T., "Tokio: logic programming language based on temporal logic", Proc. of 3rd Int. Cong. Logic Prog., E.Shapiro(editor), Lecture Notes in Computer Science 225, Springer-Verlag, 695-709.

57. Ghezzi, C., Mandrioli, D., and Morzenti, A., "TRIO, a logic language for executable specifications of real-time systems", Journal of Systems and Software, 12(2):107-123, May 1990.

58. Glinert, E.P. and Tanimoto, S.L., "Pict : An Interactive Graphical Programming Environment", Computer, Vol.17, No.11, Nov.1984, pages 7-25.

59. GNOME website at "http://www.gnome.org/"

60. Goldblatt, R., "Logics of time and computation", CSLI Lecture Notes 7, CSLI, Stanford University, Stanford.

61. Golsen, S., "State Machine Design Techniques for Verilog and VHDL", Synopsys Journal of High Level design, September 1994.

62. Gordon, M.J.C., "LCF-LSM : A system for specifying and verifying hardware", Computer Laboratory, University of Cambridge, Technical Report, 41, 1983.

63. Gordon, M., "Mechanizing programming logics in higher order logic", Technical Report 145, Univ. of Cambridge, Cambridge, UK, 1988.

64. Guttag, J.V., and Horning, J.J., "Larch : Languages and Tools for Formal Specification", Springer-Verlag, 1993.

65. Hailpern, B.T., "Verifying Concurrent Processes Using Temporal Logic", Lecture Notes in Computer Science, 129, Springer-Verlag, Berlin.

66. Hailpern, B.T. and Owicki, S.S, "Modular verification of computer communication protocols", IEEE Transactions on Comm., COM-31, 1:56-58.

67. Halbwachs, N., "Synchronous programming of reactive systems", Kluwer Academic Publishers, 1993.

68. Harel, D., "Statecharts : a visual formalism for complex systems", Science of Computer Programming, 8(1):231-274, 1987.

69. Harel, D., and Naamad, Amnon, "The STATEMATE Semantics of Statecharts", ACM Transactions on Software Engineering Method, October 1996.

70. Hayes, I.J., Jones, C.B., "Specifications are not (necessarily) executable", Software Engineering Journal, 1989, 4, (6), pages 330-338.

71. Heimdahl, M.P.E., Leveson, N., "Completeness and consistency analysis in hierarchial state-based requirements.", IEEE Trans. on Software Eng. SE-22(6), June 1996, pages 363-377.

72. Heitmeyer, C., "On the Need for Practical Formal Methods", In the proceedings of the 5th International Symposium, FTRTFT'98, Lyngby, Denmark, September 1998, volume 1486 of Lecture Notes in Computer Science, pages 18-26.

73. Heitmeyer, C., Jeffords, R., and Labaw, B., "Automated consistency checking of requirements specifications.", ACM Trans. Software Eng. and Method. 5(3), 1996.

74. Heitmeyer, C., Jeffords, R., and Labaw, B., "Comparing different approaches for specifying and verifying real-time systems", In Proc. Tenth IEEE Workshop on Real-Time Operating Systems and Software, pages 122-129, New York, May 1993.

75. Henzinger, T.A., Nicollin, X., Sifakis, J., and Yovine, S., "Symbolic model checking for real-time systems", Information and Computation, 111(2):193-244, 1994.

76. He, Jifeng, "Specification oriented semantics for ProCoS programming language $PL^{time}$, Oxford University, 1991, PRG-OU-HJF-71.

77. Hoare, C.A.R., "An axiomatic basis for computer programming", Comm. ACM, 12(10), pages 576-580.

78. Hoare, C.A.R., "Communicating Sequential Processes", Prentice-Hall, New York, 1985.

79. Hopcroft, John E., and Ullman, Jeffery, D., "Introduction to automata theory, languages, and computation", Addison-Wesley, 1980.

80. Huth, Michael R.A., Ryan, Mark D. "Logic in Computer Science - Modelling and reasoning about systems"

81. I-Logix product information at "http://www.ilogix.com".

82. Ince, Darrel, "Set Piece", "System and Software Requirements Engineering", IEEE Computer Society Press Tutorial, Edited by Richard H. Thayer and Merlin Dorfman, 1990.

83. Ince, Darrel C., "An Introduction to Discrete Mathematics and Formal System Specification", Oxford Applied Mathematics and Computing Science Series, Clarendon Press, 1988, Chapter 9 onwards.

84. ITL homepage on the internet, "http://www.cse.dmu.ac.uk/~cau/itlhomepage/index.html".

85. Jackson, M.A., "Principles of Program Design", 1975, Academic Press Inc., New York.

86. Jacky, Jonathan, "The Way of Z : Practical Programming With Formal Methods", June 1997, paperback, Cambridge University Press.

87. Jacob, Jeremy, "The Varieties of Refinement", J.M.Morris and R.C. Shaw, eds., Proceedings of the 4th BCS-FACS Refinement Workshop, Cambridge, pp9-11, Springer Verlag, Januaray 1991.

88. Jahanian, F. and Mok, A.K., "Safety analysis of timing properties in real-time systems", IEEE Trans. on Software Eng., 12(9):890-904, September, 1986.

89. Jahanian, F., and Mok, A.K., "Modechart : A specification language for real-time systems", IEEE Trans. on Software Eng., 20(10):879-889, October 1994.

90. Johnson, D., "The flow of control notations Pancode and Boxcharts", SIGPLAN Notices, 25, August, 106-119.

91. Joseph, Mathai, Editor, "Chapter 1 : Time and Real Time by Joseph, Mathai", "Real-time Systems - Specification, Verification and Analysis", Prentice Hall International(UK) Ltd, 1996.

92. Jullig, Richard and Srinivas, Y.V., "Diagrams for Software Synthesis", In proceedings of The Eighth Annual Knowledge-Based Software Engineering Conference" held in Chicago. The paper is available from "http://www.kestrel.edu".

93. Goldberg, A and Kay, Alan, "Smalltalk 72 Instruction Manual", Xerox Parc, Palo Alto, California, 1976.

94. KDE website "http://www.kde.org/"

95. Knight, John C., "Challenges in the Utilization of Formal Methods", In the proceedings of the 5th International Symposium, FTRTFT'98, Lyngby, Denmark, September 1998, volume 1486 of Lecture Notes in Computer Science, pages 1-17.

96. Korf, F. and Schlör, R., "Interface controller synthesis from requirement specifications", In Proceedings, The Europen Conference on Design Automation, pages 385-394, Paris, France, February 1994, IEEE Computer Society Press.

97. Krieg-Brückner, B., Peleska, J, Olderog, E.-R., et al., "UniForM - Universal Formal Methods Workbench", In Statusseminar des BMBF Softwaretechnologie, pages 357-378, BMBF, Berlin, 1996.

98. Kromodimoeljo, S., Pase, W., Saaltink, M., Craigen, D., and Meisels, I., "A tutorial on EVES", Technical Report, Odyssey Research Associates, Ottawa, Ont., 1993.

99. Lamport, L., "What good is temporal logic", In Proc. IFIP 9th World Congress, R.E.A.Mason(editor), North-Holland, pages 657-668.

100. Lee, I., Bremond-Gregoire, and Gerber, R., "A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems", Proceedings of the IEEE, pages 158-171, Jan. 1994.

101. Leigh, A.W., "Real Time Software for Small Systems", Published in UK by Sigma Press, Wilmslow, First Edition, 1988.

102. Liu, Xiadong, Yang, Hongji and Zedan, Hussein, "Speed and Scale Up Software Reengineering with Abstraction Patterns and Rules", In proceedings of the International Conference on Software Maintenance (ICSM) 2000, San Jose, California.

103. Liu, Xiadong, "Abstraction : A notion for reverse engineering", Ph.D. Thesis, De Monfort University, Leicester, UK, September 1999.

104. Lüth Karsten, "The ICOS Synthesis Environment", In the proceedings of the 5th International Symposium, FTRTFT'98, Lyngby, Denmark, September 1998, volume 1486 of Lecture Notes in Computer Science, pages 294-297.

105. Lowe, G. and Zedan, H., "Refinement of complex systems : a case study", The Computer Journal, 38, No.10, 1995.

106. Lynch, N. and Vandraager, F., "Forward and backward simulations for timing-based systems", In proceedings of REX workshop "Real-Time : Theory in Practice", volume 600 of LNCS, pages 397-446, Mook, The Netherlands, June 1991, Springer-Verlag.

107. Manna, Z., Pnueli, A., "The Temporal Logic of Reactive and Concurrent Systems - Specification", Springer-Verlag, New York, Inc., 1992.

108. McCarthy, J., "Towards a mathematical science of computation", In proceedings, IFIP, 1961, pages 21-28.

109. Merlin, P. and Farber, D., "Recoverability of communication protocols",, IEEE Trans. on Communications, 24(9):1036-1043, September 1976.

110. Merritt, M., Modugno, F., and Tuttle, M., "Time constrained automata", In Baeten, J.C.M. and Goote, J.F., editors, CONCUR'91 : 2nd International Conference on Concurrency Theory, volume 527 of LNCS, pages 408-423, Amsterdam, The Netherlands, August 1991, Springer-Verlag.

111. Milner, R., "Communication and Concurrency", Prentice-Hall, New York, 1985.

112. Morgan, Carroll, "Programming from Specifications", Prentice Hall International, 1990.

113. Moser, L.E., Ramakrishna, Y.S., Kutty, G., Melliar-Smith, P.M. and Dillon, L.K., "A Graphical Environment for the Design of Concurrent Real-Time Systems", ACM Transactions on Software engineering and Methodology, 6(1):31-79, 1997.

114. Moszkowski, B.C., "Reasoning about Digital Circuits", Ph.D. Thesis, Stanford University Technical Report STAN-CS-83-970.

115. Moszkowski, B.C., "Compositional Reasoning using Interval Temporal Logic and Tempura", LNCS, Vol. 1536, pp 0439-.

116. Moszkowski, Ben, "Executing Temporal Logic Programs", Cambridge University Press, 1986. An online version of the book is available from "http://www.cse.dmu.ac.uk/~cau/itlhomepage/node2.html".

117. Moszkowski, Ben, "An Automata-Theoretic Completeness Proof for Interval Temporal Logic (Extended Abstract)", In Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP 2000), editors : Ugo Montanari, Jose Rolim and Emo Welzl, Geneva, Switzerland, July 9-15, 2000. LNCS, vol.1853, Springer-Verlag, pages 223-234.

118. Moszkowski, Ben, "A Complete Axiomatization of Interval Temporal Logic with Infinite Time (Extended Abstract)", In Proceedings of the Fifteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2000), June 26-29, 2000, Santa Barbara, California, USA, IEEE Computer Society Press, pages 242-251.

119. Myers, B.A., "Taxonomies of visual programming and program visualization", Journal of Visual Languages and Computing, 1(1):97-123, 1990.

120. Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I : Planning and Technology Insertion", produced by Office of Safety and Mission Assurance, National Aeronautics and Space Administration (NASA), NASA-GB-002-95, Release 1.0, July 1995.

121. Ostroff, J., "Temporal Logic for Real-Time Systems", Research Studies Press LTD., Taunton, Somerset, England, 1989.

122. Ostroff, J., "Visual tools for verifying real-time systems", In Rus, T. and Rattray, C., editors, Theories and Experiences for Real-Time System Development, AMAST Series in Computing, Volume 2, pages 83-101, Singapore, 1994, World Scientific Publishing Co..

123. Owicki, S. and Lamport, L., "Proving livenes properties of concurrent programs", ACM Transactions on Programming Languages and Systems, 4(3), pages 455-495.

124. Owre, S., Shankar, N., and Rushby, N., "User guide for the PVS specification and verification system(draft)", Technical Report, Comp. Sci. Lab., SRI Intl., Menlo Park, CA, 1993.

125. Peterson, J.L., "Petri Net Theory and Modeling of Systems", Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

126. Pnueli, A., "Applications of temporal logic to the specification and verification of reactive systems : A survey of current trends", Current Trends in Concurrency, Bakker, J.W.de, Roever, W.P.de and Rozenberg, G.(editors), Lecture Notes in Computer Science 224, Springer-Verlag, Berlin, pages 510-584.

127. Pnueli, A., "The temporal logic of programs", Proc. 18th IEEE Symposium on Foundations of Computer Science, pages 46-57.

128. Pnueli, A., "The temporal semantics of concurrent programs", Theoretical Computer Science, 13, pages 1-20.

129. Pressman, R.S., "Software Engineering - A Practitioner's approach", Third edition, McGraw Hill, Inc., International edition 1992.

130. Puigsegur i Figueras, Jordi and Agusti i Cullell, Jaume, "Using a Visual Syntax in Logic Programming" Technical Report IIIA 96-2, Institut d'Investigacio en Intel-ligencia Artificial, Bellaterra, Catalonia, March 1996.

131. PVS homepage on the internet, "http://www.csl.sri.com/pvs.html".

132. Reed, G. and Roscoe, A., "Metric spaces as models for real-time concurrency", In Proceedings, Mathematical Foundations of Computer Science, LNCS, volume 298, New York, 1987, Springer-Verlag.

133. Richardson, Debra J., "Modeling Automobile Cruise Control System Using Graphical Interval Logic", from "http://www.ics.uci.edu/~djr/"

134. Ruiz, Marrero, Suárez, Alvaro, "A tool for visualizing LOTOS Behavioural Specifications", 4th FTRTFT, LNCS 1135, pages 475-478, Sept. 1996.

135. Scholefield, D.J., Zedan, H. and He, J., "Real-time refinement : semantics and application", LNCS, 711, pages 693-701, 1993.

136. Scholefield, D.J., Zedan, H. and He, J., "A specification oriented semantics for the refinement of real-time systems", Theoretical Computer Science, 130, August 1994.

137. Selic, B., Gullekson, G., and Ward, P., "Real-Time Object Modeling", John Wiley, 1994.

138. Shlaer, S. and Mellor, S., "Object oriented system analysis : Modeling the world in data", Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

139. Shu, N.C., "Visual Programming Languages : A Perspective and a Dimensional Analysis", In Visual Languages, editors Chang, S.K., Ichikawa, T., and Ligomenides, P.A., pages 11-34, Plenum, New York, 1986.

140. Sinan Si Alhir,"UML in a Nutshell - A Desktop Quick Reference", O'Reilly & Associates, Inc., First Edition, September 1998.

141. Skakkebaek, J., Shankar, Natarajan, "Towards a Duration Calculus proof assistant in PVS", In Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, volume 863 of Lecture Notes in Computer Science, 1994, pages 660-679.

142. SPIN webpage for the model checker at http://netlib.bell-labs.com/netlib/spin/whatispin.html

143. "Stateflow User's Guide", The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098.

144. Tapken, Josef, "Interactive and Compilative Simulation of PLC-Automata", In W.Hahn and A.Lehmann, editors, Simulation in Industry, ESS'97, pages 552-556, SCS, 1997

145. Thomas, Wolfgang, "On the synthesis of strategies in infinite games", In E.W.Meyer and C.Puech, editors, Symposium on Theoretical Aspects of Computer Science(STACS 95), volume 900 of Lecture Notes in Computer Science, pages 1-13, Springer-Verlag, March 1995.

146. Tkpaint homepage at http://www.netanya.ac.il/~samy/tkpaint.html

147. "Visualization in Computing", A special issue, IEEE Computer 22, No.10, pp.9-65, October(1989)".

148. Ward, P.T. and Mellor, S.J., "Structured Development for Real-Time Systems", Yourdon Press, New York, 1985.

149. Ward, M., "A definition of abstraction", Technical report, Durham University, 1992.

150. Welch, Brent B., "Practical Programming in Tcl and Tk", second edition, Prentice-Hall, Inc., 1997.

151. Wing, J.M., "A specifier's introduction to formal methods", Computer, 1990, 7, (5), pages 8-24.

152. Yourdon, E., and Constantine, L.L., "Structured Design", Yourdon Press, 1979.

153. Zhou, Chaochen, Hoare, C.A.R., and Ravn, A.P, "A calculus of durations" Information Proc. Lettters, 40(5), December 1991.

154. Zhou, Chaochen, "Duration Calculi : An overview", In D.Bjömer, Broy, M., and Pottosin, I.V., editors, Proc. Formal Methods in Programming and Their Application, volume 735 of LNCS, pages 256-266, Springer-Verlag, 1993.

# Appendix A

# Temporal Agent Model

## A.1 Computational Model

A model of computation defines mathematically an abstract architecture upon which applications will execute. A system is a collection of agents (which is our unit of computation), possibly executing concurrently and communicating synchronously or asynchronously via communication links. Systems can themselves be viewed as single agents and composed into larger systems. Systems have timing constraints imposed at three levels; system wide communication deadlines, agent deadlines and sub-computation deadlines (within the computation of an individual agent). Deadlines are all considered to be hard. A system has a static configuration, and it must have at most a finite number of communication links and agents. At any instant in time, a system can be thought of as having a unique state. The system state is defined by the values in the communication links and the state variables of the system, the so-called frame. This frame defines the variables that can possibly change during system execution, the variables outside this frame will certainly not change. Computation is defined to be a sequence of system states, i.e. an interval of states. ITL enables us to describe these sets of computations in an eloquent way. An agent is described by a computation which may transform a local data-space and may read read and write to communication links during execution. The local data-space for the agent is created when the agent star ecution and is destroyed when the agent terminates. No agent may read or write another agent's local data-space. The computation may have both minimum and maximum execution times imposed. An agent may perform both computation and communication. Only an agent which performs no computation or communication may terminate instantaneously. Such an agent is called an empty agent. An agent may start execution as a result of either a condition of the current time or a write event occurring on a specific communication link.

An agent may write to at most a finite number of communication links and read from at most a finite number of them. Synchronous communication links are called channels where read and write occurs at the same time. Asynchronous communication links are called shunts. Synchronous communication links modelled by a shared variable which contains three values: the first one indicates if there is an agent willing to read from the channel, the second one if there is an agent willing to write to the channel and the third is the value transmitted over the channel. Asynchronous communication links are modelled by a shared variable which contains two values : the first one is a stamp which is increased by one each time a new write to the shunt takes place, and the second one is the value which was most recently written. Shunt writing is destructive, shunt reading is not.

Communication link readership may be restricted to a set of agents. These agents can then be considered as a subsystem where communication links which are read or written by the agents within the subsystem define the subsystem's boundary. Subsystems may not overlap. The stamps within the shunts enable the reading agents to compute according to the freshness of the data. The need for stamps in shunts is a direct consequence of the decision to use non-destructive asynchronous communication. When an agent performs two consecutive inputs from a shunt and reads the same data item twice, it may need to know if each value is a result of two different writes or a single write.

For a detailed treatment of the syntax and semantics of TAM, the reader is referred to [135, 105, 29]. The following gives, in more detail, the concrete constructs introduced in chapter 4.

**TAM concrete constructs**

This following introduces the concrete constructs for reasoning about communication, timing and resource allocation.

- **Channel communication :**

  Let $C$ be a channel then channel $C \in P$ denotes that a new channel is introduced. $C!e$ denotes an output agent that sends the value of expression $e$ over $C$. $C?x$ denotes an input agent that stores the value received over $C$ in $x$.

$$\begin{array}{lcl}
\mathsf{channel}\ C\ \mathsf{in}\ P & \hat{=} & \exists C \bullet P \\
C? & \hat{=} & \Pi_1(C) = \mathit{true} \\
C! & \hat{=} & \Pi_2(C) = \mathit{true} \\
C.x & \hat{=} & \Pi_3(C) = x \wedge C? \wedge C! \\
C!e & \hat{=} & (\neg C? \wedge C! \wedge \mathsf{stable}(C)\,;\mathsf{skip}) \vee \mathsf{empty}\,;C.e \\
C?x & \hat{=} & (\neg C! \wedge C? \wedge \mathsf{stable}(C)\,;\mathsf{skip}) \vee \mathsf{empty}\,;C.x
\end{array}$$

  $\Pi_i$ is the projection function that for $i = 1$ gives the "willing to read" value, for $i = 2$ gives the "willing to write" value and for $i = 3$ the actual value in the channel. In the first interval the agent is waiting for its partner and in the second interval communication takes place.

  Let $d \in TIME$. The notation $C!_d e$ $(C?_d x)$ specifies an agent which is willing to perform the communication at time $d$. However, the agent will be held up forever if the environment fails to react promptly.

$$\begin{array}{lcl}
C!_d e & \hat{=} & C!e \wedge (\mathit{finite} \supset \mathit{len} = d) \\
C?_d x & \hat{=} & C?x \wedge (\mathit{finite} \supset \mathit{len} = d)
\end{array}$$

- **Shunt communication :**

Let $s$ be a shunt then shunt $s$ in $P$ denotes that a new shunt $s$ is introduced. write$(v,s)$ denotes that value $v$ is written to shunt $s$, read$(s)$ gives the value stored in shunt $s$ and $\surd s$ gives the stamp of shunt $s$. These agents are defined as follows:

$$\begin{array}{lcl} \surd s & \mathrel{\widehat{=}} & \Pi_1(s) \\ \mathsf{shunt}\, s \,\mathsf{in}\, P & \mathrel{\widehat{=}} & \exists s \bullet \surd s = 0 \wedge P \\ \mathsf{write}(v,s) & \mathrel{\widehat{=}} & \mathsf{skip} \wedge \bigcirc s = (\surd s + 1, v) \\ \mathsf{read}(s) & \mathrel{\widehat{=}} & \Pi_2(s) \end{array}$$

where $\Pi_i$ is the projection function that for $i = 1$ gives the stamp and for $i = 2$ gives the value stored in the shunt.

Let $d \in Time - \{0\}$. The notation write$_d(v,s)$ specifies an agent that writes value $v$ to shunt $s$ at time $d$.

$$\mathsf{write}_d(v,s) \mathrel{\widehat{=}} len = d - 1 \,;\mathsf{skip} \wedge \bigcirc s = (\surd s + 1, v)$$

Note: the value of the stamp is defined to be the value of the stamp in the previous state plus 1.

If one wants a version of write$_d$ which remains stable except in the last state of the interval one can take pwrite$_d(v,s)$ which is defined as

$$\mathsf{pwrite}_d(v,s) \mathrel{\widehat{=}} \mathsf{write}_d(v,s) \wedge padded(s)$$

- **Delay and timeout :**

Let $d \in TIME \cup \{\infty\}$. The notation delay$_d$ describes an agent which first holds up for $d$ time units and then terminates with all global variables untouched. Its execution does not claim any resource time

$$\mathsf{delay}_d \mathrel{\widehat{=}} len = d$$

Let $d \in TIME \cup \{\infty\}$. The notation $P \trianglelefteq_d Q$ defines an agent which behaves like $P$ if $P$ is executed within $d$ time units, otherwise it behaves like agent $Q$.

$$P \trianglelefteq_d Q \mathrel{\widehat{=}} \mathsf{if}(P \supset finite \wedge len \leq d)\, \mathsf{then}\, P \,\mathsf{else}\, Q$$

- **Resource allocation :**

Let *res* be a resource then $\mathsf{request}(v, res)$ is the agent that requests $v$ units of resource *res*. If these $v$ units are not available it waits for them. $\mathsf{release}(v, res)$ is the agent that releases $v$ units of resource *res*.

$$\begin{aligned}
\mathsf{request}(v, res) &\mathrel{\widehat{=}} \mathsf{if}\ res \geq v\ \mathsf{then}\ res := res - v\ \mathsf{else}\ \bigcirc(\mathsf{request}(v, res)) \\
\mathsf{release}(v, res) &\mathrel{\widehat{=}} \bigcirc res = res + v
\end{aligned}$$

# Appendix B

# Core Tcl-Tk procedures for the VisITL tool

## B.1 Core Drawing Procedures for VisITL Constructs

The following procedure is used in the drawing of VisITL constructs like "Next", "Always", "Sometime" etc.. when the user selects a VisITL construct on the tool-bar and starts drawing on the canvas.

```
proc startFormula_enhancer {qualifier} {
  global Formula_enhancer Graphics
  global enhancer_line Graphics
  global enhancer_text Graphics
  global textstr
  set Formula_enhancer(shape) Formula_enhancer
  set enhancer_line(shape) enhancer_line
  set enhancer_text(shape) enhancer_text
  set textstr $qualifier
  bind .c <Button-1> {
       global x1 y1 x2 y2
```

```
      set x [.c canvasx %x $Graphics(grid,snap)]
      set y [.c canvasy %y $Graphics(grid,snap)]
      set x1 $x
      set y1 $y
      set Formula_enhancer(coords) "$x $y $x $y"
           set Formula_enhancer(options)  [list \
          -width   4 \
          -outline blue \
          -stipple $Graphics(fill,style) \
          -tags {Formula_enhancer obj}       \
      ]

      set enhancer_line(options) [list \
          -width 2 \
          -tags  {enhancer_line obj}   \
          -fill black
      ]
      set enhancer_text(options) [list \
          -tags {enhancer_text obj} \
          -text $textstr \
          -font {-size 15 -weight bold -slant italic} \
      ]
      }
 bind .c <B1-Motion> {
      global x1 y1 x2 y2
      set x [.c canvasx %x $Graphics(grid,snap)]
      set y [.c canvasy %y $Graphics(grid,snap)]
      makeformula_enhancer $x $y
 }
 bind .c <B1-ButtonRelease> {
      global x1 y1 x2 y2 yline Formula_enhancer enhancer_line
if ![info exists Formula_enhancer(id)] {return}
      set utag [Utag assign $Formula_enhancer(id)]
      set x2 [.c canvasx %x]
      set y2 [.c canvasy %y]
      set yline [expr ($y1 + 0.25*($y2 - $y1))]
      set x3 [expr ($x1 + 0.50*($x2-$x1))]
      set enhancer_line(id) [eval .c create line $x1 $yline $x2 \
          $yline $enhancer_line(options)]
#if ![info exists Formula_enhancer(id)] {return}
#      set utag [Utag assign $Formula_enhancer(id)]
#  .c addtag $utag closest $x3 $yline
# The "exists" part was shifted above the lines for the creation of
# the enhancer line so that a mouse click and release at the same point
# does not end up creating a line (at the same point) thereby entering
# an additional line in the .pic file. ...........12 August 1999
 .c addtag $utag withtag [eval set tag_line [Utag assign \
                                 $enhancer_line(id)]]
      set ytext [expr $yline - 5]
      set fillcolor black
  if {$textstr == "always"} {
     set fillcolor red
  } else { if {$textstr == "next"} {
           set fillcolor blue
         } else { set fillcolor black }
   }
  if {$textstr == "chopstar"} {
     set fillcolor DodgerBlue }

  set enhancer_text(id) [eval .c create text $x3 $ytext \
                         $enhancer_text(options) -fill $fillcolor]


 .c addtag $utag withtag [eval set tag_tex [Utag assign \
                          $enhancer_text(id)]]
  set taginfo [.c itemcget $Formula_enhancer(id) -tags]
# puts $taginfo
  set taginfo [.c itemcget $enhancer_line(id) -tags]
# puts $taginfo
  set taginfo [.c itemcget $enhancer_text(id) -tags]
# puts $taginfo
      History add [getObjectCommand $utag 1]
      Undo add ".c delete $utag"
      unset enhancer_text
      unset enhancer_line
      unset Formula_enhancer
  }
  Message "Found any bugs?"
}
```

The following procedure was called in the previous procedure. It completes the drawing of the VisITL construct at the point the user releases the left mouse-button.

```
proc makeformula_enhancer {x y} {
  global Formula_enhancer x1 y1 yline
  set Formula_enhancer(coords) [lreplace $Formula_enhancer(coords) \
                                 2 3 $x $y]
  catch {.c delete $Formula_enhancer(id)}
 set Formula_enhancer(id) \
      [eval .c create rectangle $Formula_enhancer(coords) \
               $Formula_enhancer(options)
]
}
###### END OF FORMULA ENHANCER SECTION
```

The following procedure is used when the VisITL construct is to be created in a specific location (as is required by refinement rules).

```
###### FORMULA ENHANCER SECTION GIVEN COORDS
proc startFormula_enhancer-with {x1 y1 x2 y2 qualifier} {
  global Formula_enhancer Graphics
  global enhancer_line Graphics
  global enhancer_text Graphics
  global textstr
  set textstr $qualifier
  set Formula_enhancer(coords) "$x1 $y1 $x2 $y2"
            set Formula_enhancer(options)  [list \
           -width   3 \
           -outline brown \
           -stipple $Graphics(fill,style) \
           -tags {Formula_enhancer obj}       \
       ]

       set enhancer_line(options) [list \
           -width 2 \
           -tags  {enhancer_line obj}   \
           -fill black
       ]
       set enhancer_text(options) [list \
           -tags {enhancer_text obj} \
          -text $textstr \
          -font {-size 15 -weight bold -slant italic} \
      ]
   makeformula_enhancer $x2 $y2
   if ![info exists Formula_enhancer(id)] {return}
      set utag [Utag assign $Formula_enhancer(id)]
      History add [getObjectCommand $utag 1]
      Undo add ".c delete $utag"
      unset Formula_enhancer
   set yline [expr ($y1 + 0.25*($y2 - $y1))]
   set enhancer_line(id) [eval .c create line $x1 $yline $x2 \
                           $yline $enhancer_line(options)]
if ![info exists enhancer_line(id)] {return}
.c addtag $utag withtag [eval set tag_line [Utag assign \
                           $enhancer_line(id)]]
set ytext [expr $yline - 5]
 History add [getObjectCommand $utag 1]
      Undo add ".c delete $utag"
      unset enhancer_line
  set fillcolor black
  if {$textstr == "always"} {
     set fillcolor red
  } else { if {$textstr == "next"} {
           set fillcolor blue
        } else { set fillcolor black }
   }
  if {$textstr == "chopstar"} {
     set fillcolor DodgerBlue }
set x3 [expr ($x1 + 0.50*($x2-$x1))]
  set enhancer_text(id) [eval .c create text $x3 $ytext \
                              $enhancer_text(options) -fill $fillcolor]
  if ![info exists enhancer_text(id)] {return}
 .c addtag $utag withtag [eval set tag_tex [Utag assign \
                               $enhancer_text(id)]]
      History add [getObjectCommand $utag 1]
      Undo add ".c delete $utag"
      unset enhancer_text
}
###### END FORMULA ENHANCER SECTION GIVEN COORDS
```

## B.2 Procedures for Conversion to Textual ITL

The following procedure is called when the user selects the menu button for converting a VisITL formula to textual ITL. The procedures it calls are given below it.

```
proc convert2ITL {} {

global Graphics Canv utagCounter TextInfo Image Zoom fileID line
global order enhancer_id ordered_boxinfo

  sort-boxes
# calls make-listof-boxes ..
  set arg_for_latex [visual2latex $ordered_boxinfo]
  write-tex $arg_for_latex
}

proc sort-boxes {} {
      global enhancer_infolist list_enhancer_coords
      global and_infolist list_and_coords num_enhancers num_and
      global plain_formula_infolist list_plain_coords num_plain
global or_infolist list_or_coords num_or
global anon_infolist list_anon_coords num_anon
global while_infolist list_while_coords num_while
global quantifier_infolist list_quan_coords num_quan
global sortedOnX IDfor chopFrom chopTo
global enhancer_textval
global boxtype ordered_boxinfo formulatext and1text and2text
global formula_is_text arg1_is_text arg2_is_text
global or1_is_text or2_is_text or1_text or2_text
global anon1_is_text anon2_is_text anon1_text anon2_text
global while1_is_text while2_is_text while1_text while2_text
global quantext quan_is_text quantifier_name quantifier_var
global coords_for_or_rect1 coords_for_or_rect2
global coords_for_or_rect3 coords_for_or_rect4
global tree1 tree2 tree3 tree4 In_Or plain_textval t onlytext

set ordered_boxinfo ""
make-listof-boxes
if {$t == 1} {
   set ordered_boxinfo onlytext
   set ordered_boxinfo [concat $ordered_boxinfo $onlytext]
} else {

set this_list [concat $list_enhancer_coords $list_and_coords \
        $list_plain_coords $list_or_coords $list_anon_coords \
        $list_while_coords $list_quan_coords]
set sortedOnX [lsort -command listCompare $this_list]
set total [llength $this_list]
set orderedIDs ""
for {set counter 0} {$counter < $total} {incr counter} {
        set a [lindex $sortedOnX $counter]
        lappend orderedIDs $IDfor($a)
      }
     for {set k 1} {$k < [expr $num_or + 1]} {incr k} {
       puts "Or info. : $or_infolist($k)"
     }
    foreach id [.c find withtag chop_line] {
       set coords_chopline [.c coords $id]
       set chopX1 [lindex $coords_chopline 0]
       set chopY1 [lindex $coords_chopline 1]
       set chopX2 [lindex $coords_chopline 2]
       set chopY2 [lindex $coords_chopline 3]
       for {set i 0} {$i < $total + 1} {incr i} {
         set currBoxCoords [lindex $sortedOnX $i]
         set boxX1 [lindex [lindex $sortedOnX $i] 0]
         set boxY1 [lindex [lindex $sortedOnX $i] 1]
         set boxX2 [lindex [lindex $sortedOnX $i] 2]
         set boxY2 [lindex [lindex $sortedOnX $i] 3]
         if {$chopX1 > $boxX1} {
          if {$chopX1 < $boxX2} {
           if {$chopY1 > $boxY1} {
            if {$chopY1 < $boxY2} {
                  set chopFrom($id) $IDfor($currBoxCoords)
            }
           }
          }
         }
         if {$chopX2 > $boxX1} {
          if {$chopX2 < $boxX2} {
           if {$chopY2 > $boxY1} {
            if {$chopY2 < $boxY2} {
                 set chopTo($id) $IDfor($currBoxCoords)
            }
           }
          }
         }
        }
        puts "chop : $id is from box : $chopFrom($id)"
        puts "chop : $id is to box : $chopTo($id)"
        set lower [lsearch $orderedIDs $chopFrom($id)]
        set higher [lsearch $orderedIDs $chopTo($id)]
        if {$lower > $higher} {
# These were swapping ; I will now change the algo -> insert just before
#  set fullyorderedIDs [lreplace $orderedIDs $lower $lower $chopTo($id)]
#  set fullyorderedIDs [lreplace $fullyorderedIDs $higher \
#                       $higher $chopFrom($id)]
# So, I will insert..
         set fullyorderedIDs [linsert $orderedIDs $higher $chopFrom($id)]
         set newlower [expr $lower + 1]
         set fullyorderedIDs [lreplace $fullyorderedIDs $newlower \
                             $newlower]
         set orderedIDs $fullyorderedIDs
         # So, I have overwritten on orderedIDs
        }
        puts "The orderedIDs after considering chops = $orderedIDs"
     }
# Added 11 May 00  -> start
     foreach id [.c find withtag Or] {
         set x1 [lindex $coords_for_or_rect1($id) 0]
         set y1 [lindex $coords_for_or_rect1($id) 1]
        set x2 [lindex $coords_for_or_rect1($id) 2]
        set y2 [lindex $coords_for_or_rect1($id) 3]
        puts "rect1 : $x1 $y1 $x2 $y2"
        foreach idee [.c find enclosed $x1 $y1 $x2 $y2] {
            set In_Or($idee) yes
            set tags [.c gettags $idee]
            puts "$idee : $tags : $In_Or($idee)"
        }
        set x1 [lindex $coords_for_or_rect2($id) 0]
        set y1 [lindex $coords_for_or_rect2($id) 1]
        set x2 [lindex $coords_for_or_rect2($id) 2]
        set y2 [lindex $coords_for_or_rect2($id) 3]
        foreach idee [.c find enclosed $x1 $y1 $x2 $y2] {
            set In_Or($idee) yes
        }
        set x1 [lindex $coords_for_or_rect3($id) 0]
        set y1 [lindex $coords_for_or_rect3($id) 1]
        set x2 [lindex $coords_for_or_rect3($id) 2]
        set y2 [lindex $coords_for_or_rect3($id) 3]
        foreach idee [.c find enclosed $x1 $y1 $x2 $y2] {
            set In_Or($idee) yes
        }
        set x1 [lindex $coords_for_or_rect4($id) 0]
        set y1 [lindex $coords_for_or_rect4($id) 1]
        set x2 [lindex $coords_for_or_rect4($id) 2]
        set y2 [lindex $coords_for_or_rect4($id) 3]
        foreach idee [.c find enclosed $x1 $y1 $x2 $y2] {
            set In_Or($idee) yes
        }
}
```

```
# 11 May 00 end
# 16 May 00  I have to use a different l for the for loop below.. which
# includes the sum of all ids. of boxes in orderedIDs which are NOT in
# any or box
       set l [llength $orderedIDs]
       set orderedIDs_notinor ""
       for {set i 0} {$i < [expr $l]} {incr i} {
           set id [lindex $orderedIDs $i]
           if {$In_Or($id) == "no"} {
              set orderedIDs_notinor [concat $orderedIDs_notinor $id]
           }
       }
       puts "The orderedIDs_notinor : $orderedIDs_notinor"
#       set l [llength $orderedIDs]
       set l [llength $orderedIDs_notinor]
       for {set i 0} {$i < [expr $l]} {incr i} {
          set thisID [lindex $orderedIDs $i]
          foreach id [.c find withtag chop_line] {
            if {[string compare $chopFrom($id) $thisID] == 0} {
               set ordered_boxinfo [concat $ordered_boxinfo chop]
            }
          }
          if {$boxtype($thisID) == 0} {
          set textval $enhancer_textval($thisID)
              if {$formula_is_text($thisID) == 1} {
                 set textval [concat $textval $formulatext($thisID)]
              }
          } elseif {$boxtype($thisID) == 1} {
              set textval and
              if {$arg1_is_text($thisID) == 1} {
```

```
            set textval [concat $textval $and1text($thisID)]
         }
         if {$arg2_is_text($thisID) ==1} {
            set textval [concat $textval $and2text($thisID)]
         }
     } elseif {$boxtype($thisID) == 2} {

         if {$In_Or($thisID) == "no"} {
        puts "NOTTTTTTTTTTTT in Or"
        set textval $plain_textval($thisID)
         }

     } elseif {$boxtype($thisID) == 3} {
         set textval or
         if {$or1_is_text($thisID) == 1} {
            set textval [concat $textval $or1_text($thisID)]
         }
         if {$or2_is_text($thisID) ==1} {
            set textval [concat $textval $or2_text($thisID)]
         }
     } elseif {$boxtype($thisID) == 4} {
         set textval anon
         if {$anon1_is_text($thisID) == 1} {
            set textval [concat $textval $anon1_text($thisID)]
         }
         if {$anon2_is_text($thisID) ==1} {
            set textval [concat $textval $anon2_text($thisID)]
         }

     } elseif {$boxtype($thisID) == 5} {
```

```
              set textval while
              if {$while1_is_text($thisID) == 1} {
                 set textval [concat $textval $while1_text($thisID)]
              }
              if {$while2_is_text($thisID) ==1} {
                 set textval [concat $textval $while2_text($thisID)]
              }
         } elseif {$boxtype($thisID) == 6} {
            set textval $quantifier_name($thisID)
            set textval [concat $textval $quantifier_var($thisID)]
               if {$quan_is_text($thisID) == 1} {
                 set textval [concat $textval $quantext($thisID)]
              }
         } elseif {$boxtype($thisID) == 7} {
            set textval or
             set textval [concat $textval orarg1 $tree1($thisID)]
             set textval [concat $textval orarg2 $tree2($thisID)]
             set textval [concat $textval orarg3 $tree3($thisID)]
             set textval [concat $textval orarg4 $tree4($thisID)]
         }
          set ordered_boxinfo [concat $ordered_boxinfo $textval]


    }


#          set ordered_boxinfo [concat $ordered_boxinfo $textval]
    puts "Finally, the order of IDs = $orderedIDs"
    }
puts "The ordered info. translates to : $ordered_boxinfo"
# NOW the ordered_boxinfo could be bracketed nicely as a tree
# going thro' it in reverse order
```

```
}
```

The following procedure is too long to be included here. The source code in STRL could be referred to for further information. This procedure makes a list of all items on the canvas (in terms of various VisITL constructs and orders it based on co-ordinates). The sort-boxes procedure does further re-ordering based on different possibilities of having the chop construct.

```
proc make-listof-boxes {} {
# please refer to source code in STRL if interested in details.
}
```

The following procedure does bracketing of the expression supplied according to the number of arguments expected for each construct. The expression supplied would be correct in accordance with VisITL semantics if procedure sort-boxes{} was called before this.

```
proc visual2latex {supplied} {

    set len [llength $supplied]
    set resultant ""
    for {set i [expr $len - 1]} {$i > -1} {incr i -1} {
       set Lb "{"
       set Rb "}"
       set str [lindex $supplied $i]
       switch -exact -- $str {

          next     { set arg1 [lindex $resultant 0] ;
                     set currLen [llength $resultant];
                     set theRest [lreplace $resultant 0 0];
```

```
                  set resultant [concat $Lb \\Next $Lb $arg1 $Rb \
                      $Rb $Lb $theRest $Rb]}
      always    { set arg1 [lindex $resultant 0] ;
                  set currLen [llength $resultant];
                  set theRest [lreplace $resultant 0 0];
                  set resultant [concat $Lb \\Always $Lb $arg1 $Rb \
                      $Rb $Lb $theRest $Rb]}
      chopstar      { set arg1 [lindex $resultant 0] ;
                  set currLen [llength $resultant];
                  set theRest [lreplace $resultant 0 0];
                  set resultant [concat $Lb \\Chopstar $Lb $arg1 \
                      $Rb $Rb $Lb $theRest $Rb]}
      sometimes   { set arg1 [lindex $resultant 0] ;
                  set currLen [llength $resultant];
                  set theRest [lreplace $resultant 0 0];
                  set resultant [concat $Lb \\Sometime $Lb $arg1 \
                      $Rb $Rb $Lb $theRest $Rb]}
      not         { set arg1 [lindex $resultant 0] ;
                  set currLen [llength $resultant];
                  set theRest [lreplace $resultant 0 0];
                  set resultant [concat $Lb \\Not $Lb $arg1 $Rb \
                                 $Rb $Lb $theRest $Rb]}
      and       { set arg1 [lindex $resultant 0] ;
                  set arg2 [lindex $resultant 1] ;
                  puts "resultant now : $resultant"
                  set currLen [llength $resultant];
                  set theRest [lreplace $resultant 0 1];
                  set resultant [concat $Lb \\And $Lb $arg1 $Rb \
                                 $Lb $arg2 $Rb $Rb $theRest] }
     orarg1       {
```

```
                set or_arg1 [lindex $resultant 0];
                set theRest [lreplace $resultant 0 0];
                set resultant $theRest
              }
      orarg2   {
                set or_arg2 [lindex $resultant 0];
                set theRest [lreplace $resultant 0 0];
                set resultant $theRest
              }
     orarg3   {
                set or_arg3 [lindex $resultant 0];
                set theRest [lreplace $resultant 0 0];
                set resultant $theRest
              }
      orarg4   {
                set or_arg4 [lindex $resultant 0];
                set theRest [lreplace $resultant 0 0];
                set resultant $theRest
              }
      or      {
                puts "resultant now : $resultant"
                set currLen [llength $resultant];
                set theRest [lreplace $theRest 0 0];
                set resultant [concat $Lb \\Or $Lb $or_arg1 $Rb \
                 $Lb $or_arg2 $Rb $Lb $or_arg3 $Rb $Lb $or_arg4 \
                 $Rb $Rb $theRest]
              }
     anon    {  set anon1 [lindex $resultant 0] ;
                set anon2 [lindex $resultant 1] ;
```

```
                    set currLen [llength $resultant];
                    set theRest [lreplace $resultant 0 1];
                    set resultant [concat $Lb \\Desc $Lb $anon1 $Rb \
                        $Lb $anon2 $Rb $Rb $theRest] }
     while     {    set while1 [lindex $resultant 0] ;
                    set while2 [lindex $resultant 1] ;
                    set currLen [llength $resultant];
                    set theRest [lreplace $resultant 0 1];
                    set resultant [concat $Lb \\While $Lb $while1 \
                    $Rb $Lb $while2 $Rb $Rb $theRest] }
       chop       { set arg1 [lindex $resultant 0] ;
                    set arg2 [lindex $resultant 1] ;
                    set currLen [llength $resultant];
                    set theRest [lreplace $resultant 0 1];
                    set resultant [concat $Lb \\Chop $Lb $arg1 $Rb \
                        $Lb $arg2 $Rb $Rb $theRest] }
       forall  {    set arg1 [lindex $resultant 0] ;
                    set arg2 [lindex $resultant 1] ;
                    set currLen [llength $resultant];
                    set theRest [lreplace $resultant 0 1];
                    set resultant [concat $Lb \\Forall $Lb $arg1 $Rb \
                        $Lb $arg2 $Rb $Rb $theRest] }
       exists  {    set arg1 [lindex $resultant 0] ;
                    set arg2 [lindex $resultant 1] ;
                    set currLen [llength $resultant];
                    set theRest [lreplace $resultant 0 1];
                    set resultant [concat $Lb \\Exists $Lb $arg1 $Rb \
                                  $Lb $arg2 $Rb $Rb $theRest] }
      onlytext { set arg1 [lindex $resultant 0] ;
                 set theRest [lreplace $resultant 0 0];
```

```
                    set resultant [concat $Lb \\Onlytext $Lb $arg1 \
                       $Rb $Rb $theRest]}

          default { set resultant [concat $Lb $str $Rb $resultant]}
       }
    }
       return $resultant
  }

proc write-tex {arg_for_latex} {

########## FILE OPERATIONS
 set fileID [open /export/home0/users/pub/arun/visual/program/ \
           test_tex/testing.tex w 0600]
 puts $fileID "\\documentclass\[12pt,a4paper\]{report}"
 puts $fileID "\\usepackage{latexsym,amssymb,times}"
 puts $fileID "\\usepackage\[dvips,final\]{graphicx}"
 puts $fileID "\\paperwidth  597.50787pt"
 puts $fileID "\\paperheight 845.04684pt"
 puts $fileID "\\textwidth  416.83289pt"
 puts $fileID "\\textheight 670.50687pt"
 puts $fileID "\\oddsidemargin  18.0675pt"
 puts $fileID "\\evensidemargin 18.0675pt"
 puts $fileID "\\topmargin  0.0pt"
 puts $fileID "\\headheight 0.0pt"
 puts $fileID "\\headsep    0.0pt"
 puts $fileID "\\footskip   30.0pt"
 puts $fileID "\\hoffset 0.0pt"
 puts $fileID "\\voffset 0.0pt"
 puts $fileID "\\newcommand{\\Always}\[1\]{\\Box \[#1\]}"
 puts $fileID "\\newcommand{\\Sometime}\[1\]{\\Diamond \[#1\]}"
 puts $fileID "\\newcommand{\\Not}\[1\]{\\neg \[#1\]}"
 puts $fileID "\\newcommand{\\And}\[2\]{\[#1\] \\mathrel{\\scriptstyle \
               \\wedge}\[#2\]}"
 puts $fileID "\\newcommand{\\Or}\[4\]{\[#1\] \\mathrel{\\scriptstyle \
              \\vee} \[#2\] \\mathrel{\\scriptstyle\\vee} \[#3\] \
              \\mathrel{\\scriptstyle\\vee} \[#4\]}"
 puts $fileID "\\newcommand{\\Equiv}{\\quad\\equiv\\quad}"
 puts $fileID "\\newcommand{\\desc}{\\mathord{\\imath}}"
 puts $fileID "\\newcommand{\\Desc}\[2\]{{\\mathord{\\imath} #1} \
              \\colon #2}"
 puts $fileID "\\newcommand{\\Nextsym}{\\raise 0.20em\\hbox{$\\ \
              scriptstyle \\bigcirc\\mskip -2.5mu$}}"
 puts $fileID "\\newcommand{\\Next}\[1\]{\\mathop{\\Nextsym} \[#1\]}"
 puts $fileID "\\newcommand{\\Chop}\[2\]{\[#1\] \\mathbin{;} \[#2\]}"
 puts $fileID "\\newcommand{\\Chopstar}\[1\]{\[#1\] ^*}"
# puts $fileID "\\newcommand{\\While1}{\\tempop{\\sf while}}"
# puts $fileID "\\newcommand{\\Do}{\\temprel{\\sf do}}"
# puts $fileID "\\newcommand{\\While}\[2\]{\\While1 \[#1\] \\Do \[#2\]}"
 puts $fileID "\\newcommand{\\While}\[2\]{While \[#1\] do \[#2\]}"
 puts $fileID "\\newcommand{\\Onlytext}\[1\]{\[#1\]}"
 puts $fileID "\\newcommand{\\Forall}\[2\]{\\forall \[#1\] \\colon \
                          \[#2\]}"
 puts $fileID "\\newcommand{\\Exists}\[2\]{\\exists \[#1\] \\colon \
                          \[#2\]}"


 puts $fileID "\\begin{document}"


 puts $fileID "The following is the equivalent textual ITL formula :\\\\"
  puts -nonewline $fileID "$"
  puts -nonewline $fileID $arg_for_latex
  puts $fileID "$"

  puts $fileID "\\end\{document\}"

  close $fileID

  exec latex /export/home0/users/pub/arun/visual/program \
            /test_tex/testing.tex
  exec mv testing.dvi ../test_tex
  exec xdvi ../test_tex/testing.dvi

}
```

## B.3 Refinement-related Procedures

The following procedure refines a VisITL construct using the while-1 rule.

```
proc ref_while1 {} {
# uses visualize_in
  global While_repeat Graphics latex_arg sub_tree_info
  set ref_text(options) [list \
          -tags {result_ref_while1 obj} \
          -font {-size 15 -weight bold -slant italic} \
      ]
  foreach id [.c find withtag While_repeat] {
    set coordinates [.c coords $id]
    set next1 [getObjectAbove $id]
    set next2 [getObjectAbove $next1]
    set x1 [lindex $coordinates 0]
set y1 [lindex $coordinates 1]
set x2 [lindex $coordinates 2]
set y2 [lindex $coordinates 3]
set encl [.c find enclosed $x1 $y1 $x2 $y2]
#These will be deleted later
set y3 [expr $y1 + 0.25*($y2 - $y1)]
set x3 [expr $x1 + 0.50*($x2 - $x1)]
set next [.c find enclosed $x1 $y1 $x2 $y3]
set required_txt ""
eval lappend required_txt [getObjectOptions $next 1]
set position [lsearch $required_txt -text]
set texposn [expr $position + 1]
set text_en [lindex $required_txt $texposn]
if {$text_en == "while"} {
   puts "I should refine now !!"
   set x0 [expr $x2 - $x1]
   set y0 [expr $y2 - $y1]
   set x1and1 [expr $x1 - 1.5*$x0]
   set y1and1 [expr $y1 - 0.5*$y0]
   set x2and1 [expr $x2 + 1.5*$x0]
   set y2and1 [expr $y2 + 0.5*$y0]
   startAnd-comp-with $x1and1 $y1and1 $x2and1 $y2and1
   set x1fe1 [expr $x1 - 1.25*$x0]
   set y1fe1 [expr $y1 - 0.25*$y0]
   set x2fe1 [expr $x1 + 0.25*$x0]
   set y2fe1 [expr $y1 + 1.25*$y0]
   startFormula_enhancer-with $x1fe1 $y1fe1 $x2fe1 $y2fe1 chopstar
   set x1fe2 [expr $x1 + 0.75*$x0]
   set y1fe2 [expr $y1 - 0.25*$y0]
   set x2fe2 [expr $x1 + 2.25*$x0]
set y2fe2 [expr $y1 + 1.25*$y0]
startFormula_enhancer-with $x1fe2 $y1fe2 $x2fe2 $y2fe2 fin
set x1fe3 [expr $x1 + $x0]
set y1fe3 [expr $y1 + 0.125*$y0]
set x2fe3 [expr $x1 + 2*$x0]
set y2fe3 [expr $y1 + 1.125*$y0]
startFormula_enhancer-with $x1fe3 $y1fe3 $x2fe3 $y2fe3 not
set x1and2 [expr $x1 - $x0]
set y1and2 [expr $y1 + 0.125*$y0]
set x2and2 [expr $x1]
set y2and2 [expr $y1 + 1.125*$y0]
startAnd-comp-with $x1and2 $y1and2 $x2and2 $y2and2
set xand [expr 0.5*($x1and2 + $x2and2)]

sub-ITLformula-in $x1 $y3 $x3 $y2


puts "x1y3x3y2 => $sub_tree_info"
set y1fe3p [expr $y1fe3 + 0.25*($y2fe3 - $y1fe3)]


visualize_in $x1fe3 $y1fe3p $x2fe3 $y2fe3 $sub_tree_info


set xand-2 [expr $x1and2 + 0.5*($x2and2 - $x1and2)]


visualize_in $x1and2 $y1and2 $xand-2 $y2and2 $sub_tree_info


sub-ITLformula-in $x3 $y3 $x2 $y2
puts "x3y3x2y2 => $sub_tree_info"


visualize_in $xand-2 $y1and2 $x2and2 $y2and2 $sub_tree_info
```

```
            foreach item $encl {
               .c delete $item
            }
            .c delete $id
            set alltags [.c find all]
            set total [llength $alltags]
            for {set i 0} {$i < [expr $total]}  {incr i} {
               set itemid [lindex $alltags $i]
               set itemtag [.c gettags $itemid]
            .c addtag result_ref_while1 withtag $itemid

            }
           set z 0.50
           Zoom $z
        }
     }
}
```

The following procedure finds the equivalent sub-ITL formula in a given region of the canvas. It is too long to be included here.

```
proc sub-ITLformula-in {xA yA xB yB} {
# please refer to the source code at STRL for additional details.
}

###### Proc. to DRAW A VIS-ITL given textual ITL in tree form

proc visualize_in {x1 y1 x2 y2 tree_info} {

   set current 0
set prev 0
set prev-of-prev 0
set total_and 0
set total_fe  0
set total_txt 0
set length [llength $tree_info]
for {set i [expr $length - 1]} {$i > -1} {incr i -1} {
   set arg [lindex $tree_info $i]
      switch -exact -- $arg {
      and         { set total_and [expr $total_and + 1] }
      next        { set total_fe [expr $total_fe + 1] }
      sometimes   { set total_fe [expr $total_fe + 1] }
      always      { set total_fe [expr $total_fe + 1] }
      chopstar    { set total_fe [expr $total_fe + 1] }
      not         { set total_fe [expr $total_fe + 1] }
      onlytext    {}
      default     { set total_txt [expr $total_txt + 1] }
    }
}
puts "Details in tree_info :"
set total_boxes_to_vis [expr $total_and + $total_fe]
set x0 [expr $x2 -$x1]
set y0 [expr $y2 - $y1]
set cx [expr 0.5*($x1 + $x2)]
set cy [expr 0.5*($y1 + $y2)]
for {set i 1} {$i < [expr $total_boxes_to_vis + 1]} {incr i} {
  set xx0($i) [expr $x0 * 0.9* pow(0.7, [expr \
               $total_boxes_to_vis -$i])]
  set yy0($i) [expr $y0* 0.9* pow(0.7, [expr \
               $total_boxes_to_vis -$i])]
```

```
}
set box_count 0
for {set i [expr $length - 1]} {$i > -1} {incr i -1} {
 set arg [lindex $tree_info $i]
 puts "arg $i : = $arg"
  switch -exact -- $arg {
     next {set current 3; puts "$i : $arg -> $current";
   incr box_count
   set xfactor [expr 0.5* ($xx0($box_count))]
   set yfactor [expr 0.5* ($yy0($box_count))]
   set ax [expr $cx - $xfactor]
   set ay [expr $cy - $yfactor]
   set bx [expr $cx + $xfactor]
   set by [expr $cy + $yfactor]
   startFormula_enhancer-with $ax $ay $bx $by next
        set prev-of-prev $prev ;
          set prev 3
    }
     sometimes {set current 3; puts "$i : $arg -> $current";
       incr box_count
       set xfactor [expr 0.5* ($xx0($box_count))]
       set yfactor [expr 0.5* ($yy0($box_count))]
       set ax [expr $cx - $xfactor]
       set ay [expr $cy - $yfactor]
       set bx [expr $cx + $xfactor]
       set by [expr $cy + $yfactor]
       startFormula_enhancer-with $ax $ay $bx $by sometimes
          set prev-of-prev $prev ;
          set prev 3
          }
```

```
always {set current 3; puts "$i : $arg -> $current";
   incr box_count
   set xfactor [expr 0.5* ($xx0($box_count))]
   set yfactor [expr 0.5* ($yy0($box_count))]
   set ax [expr $cx - $xfactor]
   set ay [expr $cy - $yfactor]
   set bx [expr $cx + $xfactor]
   set by [expr $cy + $yfactor]
   startFormula_enhancer-with $ax $ay $bx $by always
   set prev-of-prev $prev ;
   set prev 3
      }
not {set current 3; puts "$i : $arg -> $current";
      incr box_count
      set xfactor [expr 0.5* ($xx0($box_count))]
      set yfactor [expr 0.5* ($yy0($box_count))]
      set ax [expr $cx - $xfactor]
      set ay [expr $cy - $yfactor]
      set bx [expr $cx + $xfactor]
      set by [expr $cy + $yfactor]
      startFormula_enhancer-with $ax $ay $bx $by not
      set prev-of-prev $prev ;
      set prev 3
      }
chopstar {set current 3; puts "$i : $arg -> $current";
      incr box_count
      set xfactor [expr 0.5* ($xx0($box_count))]
      set yfactor [expr 0.5* ($yy0($box_count))]
      set ax [expr $cx - $xfactor]
      set ay [expr $cy - $yfactor]
```

```
            set bx [expr $cx + $xfactor]
            set by [expr $cy + $yfactor]
            startFormula_enhancer-with $ax $ay $bx $by chopstar
            set prev-of-prev $prev ;
            set prev 3
            }
        and {set current 2; puts "$i : $arg -> $current";
             set xfactor [expr 0.5* ($xx0($box_count))]
             set yfactor [expr 0.5* ($yy0($box_count))]
             set ax [expr $cx - $xfactor]
             set ay [expr $cy - $yfactor]
             set bx [expr $cx + $xfactor]
             set by [expr $cy + $yfactor]

             startAnd-comp-with $ax $ay $bx $by ;
             set prev-of-prev $prev ;
             set prev 2
             }
        onlytext {
                }
        default {set current 1; puts "$i: $arg -> $current";
                 set delta 25;
                 set xtext [expr 0.5*($x1 + $x2)] ;
                 set ytext [expr 0.5*($y1 + $y2)] ;
                 if {$prev == 1} {
                 set xtext [expr $xtext + $delta]
                 set ytext [expr $ytext + $delta]
                 } ;
                 set texid [eval .c create text $xtext $ytext -text \
                            $arg -tag obj] ;
```

```
                    set prev-of-prev $prev ;
                    set prev 1
                   }


     }
}
}
```