INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

REASONING SYSTEM FOR REAL TIME REACTIVE SYSTEMS

Ghayath Haidar

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

December 1999

Abstract

Reasoning system for real time reactive systems

Ghayath Haidar

Real time reactive systems are complex systems that react with their environment through stimulus response behaviour. TROMLAB development environment is a formal system being developed at Concordia University. It is the basis of the real time reactive system that will be described in this thesis.

One of the main uses of the simulation tool is debugging. The *Reasoning System* is a very good complement of the simulation tool.

The scope of this thesis is the study of a *Reasoning System* that can be used along with the simulation tool to help debug the design and verify system properties during the development phase in TROMLAB environment.

To my wife and my children.

Acknowledgments

I would like to thank my wife who supported me during these last three years without her help this would have been impossible. I would also like to thank my children and hope to compensate them for all the time we did not spend together. I would like to thank my professor **Dr. V.S. Alagar**. He was very helpful and always available, without him this thesis would not be possible. He guided me and helped me a lot. I will never forget all the exciting debates we had during this work. I would also like to thank D.Muthiayen for his big contribution in the discussions we had. I would also like to thank V. Srinivasan for all the help he offered, and all the "white nights" we spent together during this work.

I would also like to thank my parents:

"slamat"

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Research in real-time reactive systems revolve around languages and methods for specification and design, methodologies for verification and validation, and development of tools and user interfaces for hiding the complexity of rigorous formalisms. This thesis addresses the issue of simulated debugging and reasoning, an important part of a validation technique for real-time reactive systems. The *Reasoning System* is integrated with the *Simulator* of TROMLAB, a rigorous real-time reactive systems development environment being built in the Department of Computer Science, Concordia University.

1.1 Real Time Reactive Systems

Reactive systems maintain a continuous ongoing interaction with their environment. Such systems are largely event driven, interact intensively with the environment through stimulus-response behaviour, and are regulated by strict timing constraints. Further these systems might also consist of both physical components and software components controlling the physical devices in a continuous manner. Although reactive systems are interactive systems, there is a fundamental difference between these two systems. Whereas both environment and processes have synchronisation abilities in interactive systems, a process in a reactive system is solely responsible for the synchronisation with its environment. That is, a process in a reactive system is fast enough to react to stimulus from the environment, and the time between stimulus and response is acceptable enough for the dynamics of the environment to be

receptive to the response. For example, a human-computer interface is an interactive system, whereas a controller for a collision-free coordinated motion of autonomous robots is clearly reactive. In the case of real-time reactive systems, stimulus-response behaviour is also regulated by timing constraints and the major design issue is one performance. Examples of real-time reactive systems include telephony, air traffic control systems, nuclear power reactors, and avionics.

Several factors contribute to the complexity of real time reactive systems. They are :

- *largeness:* telephony and air traffic control systems are made up of a large number of components;
- *time constraints:* telephony imposes *soft* time constraints, a violation of which may not cause any catastrophe; however, avionics and nuclear power reactor control systems impose *hard* (strict) time constraints, which if violated will cause damage and injury to human safety;
- *criticality:* nuclear power reactor controller is a safety-critical system;
- *heterogeneity:* sensors, actuators, and system processes have different functional and time sensitive capabilities.

The *reactive behaviour* of the system is a combination of its functional behaviour, causal dependencies of actions, and real-time durations governing them. Due to these three layers of interaction, understanding or reasoning about the behaviour of real-time reactive systems becomes difficult. In TROMLAB, these are resolved through the introduction of the following steps:

- appropriate formalisms for specification and design refinement,
- process model support for iterative design, animated analysis, and design-time debugging,
- browser support for active reuse of design and specification artifacts, and
- an integrated GUI supporting all the above features.

1.2 Scope of the Thesis

In designing TROMLAB environment we have been motivated by the need for rigorous development methods for real-time reactive systems. The class diagrams, state machine representations, and the subsystem configuration have formal syntax and semantics. Before committing a system design to its implementation, the process model in TROMLAB requires the modelled system to be validated and verified for the satisfaction of system properties. The central piece of the *Animator* is a simulator which simulates the specified system strictly according to the specification. The current thesis is an important contribution to the simulated debugging and reasoning within the animator.

The thesis briefly reviews the architecture and design components of TROMLAB environment in Chapter 2. The planning stage of the*Reasoning System* identified the twin needs: the ability to integrate with the *Simulator* and the *GUI*. Since the design of *GUI* forced a complete re-engineering of the *Interpreter*, and the *Animator*, we discuss the design of these components in Chapters 3, and 4. Having defined the context in which the *Reasoning System* is to function, we discuss a set of requirements of the *Reasoning System* in Chapter 5. In Chapter 6, we give descriptions of the algorithms for processing different queries, and comment on their complexities. Chapter 7 gives a dialogue of the *Reasoning System*, implemented in Java, for *Robotics Assembly* case study. Chapter 8 concludes the thesis with a summary of the contributions and possible extensions to the reasoning system.

Chapter 2

TROMLAB Environment - a brief review of initial design

The TROMLAB environment is an integrated facility based on TROM formalism [Ach95] and built around a process model that incorporates iterative development, incremental design, and application of formalism through the different stages of design. The process model incorporates an iterative development approach, the benefits of which are well-known for:

- reducing risks by exposing them early in the development process,
- giving importance to the architecture of the software unit, and
- designing modules for large scale software reuse.

The TROMLAB environment provides facilities for modular design of TROM classes, modular composition of objects to build subsystems and analyse system capabilities through simulation and verification [Mut96]. An *Interpreter* and *Animator* were the first components to be built. Recently, a *Browser* has been added. In conjunction with the current effort, the following components have been built:

1. *Reasoning system:-* to aid simulated debugging and reasoning of systems during development;
2. *PVS axiom generator:-* a tool based on the verification methodology given in [Pom99] to generate axiomatic descriptions of specified classes and subsystems in PVS;

3. *Mechanised verifier:-* a verification assistant which can be used to prove safety properties of the system stated as lemmas in PVS theories.

4. *Graphical User Interface:-* to provide a comprehensive interface to all the above stages of reactive systems development.

2.1 TROMLAB Formalism

The three tier structure of the object oriented methodology introduced by [Ach95], as shown in Figure 1, is the basis of TROMLAB environment for developing reactive systems. The benefits derived from the object oriented techniques include modularity and reuse, encapsulation, and hierarchical decomposition using inheritance. In this methodology, the system requirement is specified in temporal logic. The system is modelled using a three tier design language.

The three tiers independently specify the system configuration, reactive classes, and the abstract data types included in reactive class definitions. Lower-tier specifications are imported into upper tiers. TROM is a hierarchical finite state machine augmented with ports, attributes, logical assertions on the attributes, and time constraints. The middle-tier formalism specifies TROM classes. Abstract data types are specified as LSL(Larch Shared Language) traits in the lowest tier, and can be used by objects modelled by TROM. The upper-most tier specifies object collaboration where each object is an instance of a TROM.

The three tiers are briefly described in the following three subsections.

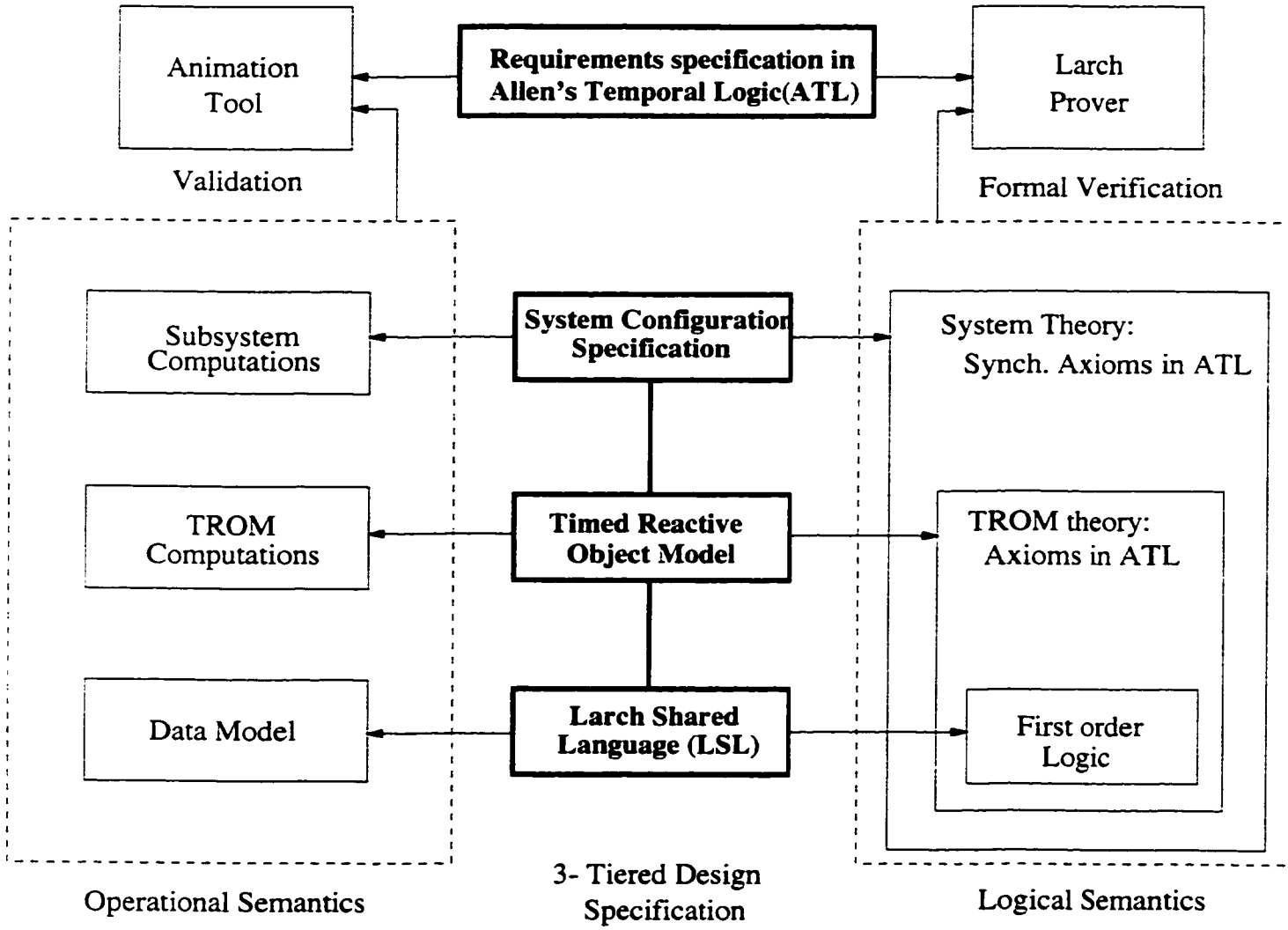


Figure 1: Three tier

2.1.1 Data Abstraction Tier

This level specifies the abstract data types included in the class definition of the middle tier. An abstract data type is defined as Larch Shared Language (LSL) trait. Larch provides a two tier approach to specification:

- First tier, called Larch Interface Language (LIL), is used to describe the semantics of a program module.
- Second tier, called Larch Shared Language (LSL), is used to specify mathematical abstractions which can be referred to in any LIL specification.

In the present implementation of TROMLAB, only LSL traits are included. An LSL trait for set data type is shown in Figure 2.

```
Trait: Set(e, S)
    Includes: Integer, Boolean
    Introduce:
        creat  :     -> S;
        insert : e, S -> S;
        delete : e, S -> S;
        size   : S    -> Int;
        member : e, S -> Bool;
        isEmpty : S   -> Bool;
        belongto: e, S -> Bool;
end
```

Figure 2: Set trait

2.1.2 TROM Tier

A TROM models a *Generic Reactive Class* (GRC). A GRC is an augmented finite state machine with port types, attributes, hierarchical states, events triggering transitions and future events constrained by strict time bounds. A state is an abstraction denoting an environment information or a system information during a certain interval of time. An event denotes an instantaneous signal. The events are classified into three types: *input*, *output*, and *internal*. Input (output) events occur at the ports of a TROM, synchronising with the output (input) events of another TROM in

its environment. The ports are abstraction of synchronous communication between TROMs. TROM objects can only interact through the port linking them as defined SCS. Only compatible ports can be linked, such that event sent at one port is acceptable as an input event at the other port at the same time [Ach95]. The specification of a transition states the conditions under which an event may occur, and the consequences of such an occurrence. The time constraints enumerate the events triggered by a transition and the time bounds within which such events should occur. Thus, a GRC is a class parameterised with port types, and encapsulates the behaviour of all TROM objects that can be instantiated from it. A formal definition of TROM is given in [Ach95].

The occurrence of an event e at a port p at time t triggers an activity which may take a finite amount of time to complete. These events may lead the TROM(s) affected by the event to undergo a state change and may further lead to the occurrence of new events as specified by the timing constraints.

2.1.3 Subsystem Specification Tier

This level is the top most tier which constitutes subsystem configuration specifications (SCS). We define the number of ports for each port type parameter in a GRC to create an object of that GRC. As in OO paradigm, several objects can be created from one GRC. These objects may have different number of ports for each port type, and consequently have the ability to communicate and interact differently with their environment. We can also include other subsystem configurations in defining a subsystem.

2.2 Syntax and Semantics

The structure and behaviour of TROM can be described either textually or visually. The templates for textual descriptions of TROMs and subsystems are shown in Figure 3, Figure 4, Figure 5, and Figure 6.

The visual representation of a reactive system includes the class diagrams, state machine diagrams, and the collaboration diagrams. These are discussed in Chapter .

```
Class Train [@C]
    Events: Near!@C, Out, Exit!@C, In
    States: *idle, cross, leave{*l1,l2}, toCross
    Attributes: cr:@C
    Traits:
    Attribute-Function: idle -> {};cross -> {};leave -> {};toCross -> {cr};
    Transition-Specifications:
        R1: <idle,toCross>; Near(true); true => !cr'=pid;
        R2: <cross,leave>; Out; true => true;
        R3: <leave,idle>; Exit(!pid=cr); true => true;
        R4: <toCross,cross>; In; true => true;
    Time-Constraints:
        TCvar2: R1, Exit, [0, 6), {};
        TCvar1: R1, In, (2, 4), {};
end
```

Figure 3: Train class specifications

```
Class Gate [ @S]
    Events: Lower?@S, Down, Up, Raise?@S
    States: *opened, toClose, toOpen, closed
    Attributes:
    Traits:
    Attribute-Function:
    Transition-Specifications:
        R1: <opened,toClose>; Lower(true); true => true;
        R2: <toClose,closed>; Down; true => true;
        R3: <toOpen,opened>; Up; true => true;
        R4: <closed,toOpen>; Raise(true); true => true;
    Time-Constraints:
        TCvar1: R1, Down, [0, 1], {closed}
        TCvar2: R4, Up, [1, 2], {};
end
```

Figure 4: Gate class specifications

```
Class Controller [ @P, @G]
    Events: Lower!@G, Near?@P, Raise!@G, Exit?@P
    States: *idle, activate, deactivate, monitor
    Attributes: inSet:PSet
    Traits: Set[@P,PSet]
    Attribute-Function: activate -> {inSet};deactivate -> {inSet};monitor -> {inSet};idle -> {};
    Transition-Specifications:
      R1: <activate,monitor>; Lower(true); true => true;
      R2: <activate,activate>; Near(!(member(pid,inSet))); true => inSet'=insertpid,inSet);
      R3: <deactivate,idle>; Raise(true); true => true;
      R4: <monitor,deactivate>; Exit(member(pid,inSet)); size(inSet)=1 => inSet'delete(pid,inSet);
      R5: <monitor,monitor>; Near(!(member(pid,inSet))); true => inSet'=insert(pd,inSet);
      R6: <monitor,monitor>; Exit(member(pid,inSet)); size(inSet)>1 => inSet'=deete(pid,inSet);
      R7: <idle,activate>; Near(true); true => inSet'=insert(pid,delete(pid, inSt));
    Time-Constraints:
      TCvar1: R7, Lower, [0, 1], {};
      TCvar2: R4, Raise, [0, 1], {};
end
```

Figure 5: Controller class specifications

```
SCS TCG
    Includes:
    Instantiate:
        t1::Train[@C:2];
        t2::Train[@C:2];
        t3::Train[@C:2];
        c1::Controller[@P:3,@G:1];
        c2::Controller[@P:3,@G:1];
        g1::Gate[@S:1];
        g2::Gate[@S:1];
    Configure:
        t1.@C1:@C <-> c1.@P1:@P;
        t1.@C2:@C <-> c2.@P1:@P;
        t2.@C1:@C <-> c1.@P2:@P;
        t2.@C2:@C <-> c2.@P2:@P;
        t3.@C1:@C <-> c1.@P3:@P;
        t3.@C2:@C <-> c2.@P3:@P;
        c1.@G1:@G <-> g1.@S1:@S;
        c2.@G1:@G <-> g2.@S1:@S;
end
```

Figure 6: SCS

The TROM model incorporates the essential features for describing reactive entities. A TROM object has a single thread control and communicates with its environment through ports by synchronous message passing. The ports represent access points for by directional communication between the objects. A port type determines the messages that are allowed at a port of that type. A TROM can have several port types associated with it and several ports of the same port type. An event represents an instantaneous activity, while an action represents an activity taking a non-atomic time interval of finite duration. At any instant, a TROM exhibits a signal representing a message, an internal activity, or idleness. The signal describes the occurrence of an event at the specific time instant, at a specific port.

Informally, the templates in Figure 3, Figure 4, and Figure 5 have the following elements:

- A set of events partitioned in three sets: input, output, and internal events.
- A set of states: A state can have sub-states.
- A set of typed attributes: The attributes can be one of the following:
  - abstract data types,
  - port reference type.
- An attribute function defining the association of attributes to states.
- A set of transition specification: Each specification describes the computational step associated with the occurrence of an event. The transition specification has three assertions: a pre- and post-condition as in Hoare logic, and the port -condition specifying the port at which the event can occur.
- A set of time-constraints: Each time constraint specifies the reaction associated with a transition. A reaction can fire an output or an internal event within a defined time period. Associated with a reaction is a set of disabling states. An enabled reaction is disabled when an object enters any of the disabling states of the reaction.

The status of a TROM captures the state in which the TROM is at that instant, the value of the attributes at the instant as reflected in the assignment vector, and the

timing behaviour of TROM as specified in the reaction vector. The reaction vector associates the set of reaction windows with each time constraint, where a reaction window represents a outstanding timing requirement to be satisfied by the output event or the internal event associated with the time constraint. When the reaction vector is null the TROM is in a stable status.

The occurrence of an activity stipulated by an interaction with the environment, or by an internal transition leads to a change in the status of a TROM. The current state of a TROM, its assignment vector, and its reaction vector can only be modified by an incoming message, by an outgoing message, or an internal signal. The status of a TROM is thus encapsulated, and cannot be modified in any other way.

A computational step [Ach95] of a TROM is an atomic step which takes the TROM from one status to its succeeding status as defined by the transition specifications. Every computational step of a TROM is associated with the transition of the TROM and every transition with either an interaction signal or an internal signal or a silent signal. The computational step occurs when the TROM receives a signal and there exists a transition specification such that the following conditions are satisfied: the triggering event for the transition is the event causing the signal; the TROM is in the source state or in a sub-state of a source state of transition specification; the port condition is satisfied if the signal is in the interaction and the enabling condition is satisfied by the assignment vector. The effects of the computational step are: the TROM enters the destination state; the assignment vector is modified to satisfy the post condition; and the reaction vector is modified to reflect the firing, disabling, and enabling of reactions. Each computational step is associated with the transition in the state machine of the TROM. After the transition is taken the current state will be the destination state of the transition. The port at which the interaction must satisfy the port condition associated with the transition, thereby constraining the objects with which the TROM can interact at that instance.

A computational step causes time-constrained responses to be activated or deactivated. If the constraint event of the outstanding reaction is the event associated with the transition, and the time of occurrence of the event associated with the transition is within the reaction window of the outstanding reaction, then the reaction is fired. If the destination state of the transition associated with the computational step is a disabling state for an outstanding reaction then the reaction is disabled. Whenever a reaction is time-constraint by the transition associated with the computational

step, the reaction is enabled. The operational semantics ensures that the time cannot advance past reaction window without either firing or disabling the associated outstanding reaction.

The factors determining whether a TROM is well formed are:

- There is at least one transition leaving every state, thus forbidding a final terminating state.
- If there is more than one transition leaving a state, then the enabling conditions of transitions should be mutually exclusive.
- Before a TROM starts executing, the value of only the active attributes in the initial state are specified. An attribute will acquire a value only when it reaches the first state in which it is active.
- Every computational step in a TROM results in some computation of the TROM.

A subsystem is composed by instantiating TROM objects from GRCs and configuring them through port links. Only compatible ports are linked between TROM objects. An already composed subsystem may also be included in composing a new subsystem: one or more of the unused ports in the objects of the included subsystem are configured with some ports of the instantiated objects in the new subsystem being composed. The objects communicate and synchronise through the configured links. The computational step of a subsystem is a vector of computational steps of the TROMs included in it.

2.3 TROMLAB Components

In this section we briefly review the functionality of the *Interpreter* [Tao96], and the *Animator*[Mut96].

2.3.1 The Interpreter

The interpreter is the first tool to be implemented in TROMLAB. The tool, as designed by Tao [Tao96], checks the textual specification for syntactic correctness and builds

an internal representation of the formal specification of a reactive system. In order to build the internal representation it performs the following tasks:

- *Syntactic analyses:* It makes sure that the files are syntactically correct; that is, consistent with TROM grammar.
- *Semantic analysis:* It does simple semantic analysis such as
  - states of a TROM have different names,
  - an LSL trait is used after being declared,
  - every transition has an outgoing and incoming state, and
  - transition specifications are well-formed logical formulas
- *Internal structure:* Based on a syntactically and semantically correct text file it generates all the internal data structures that would be used by all the other tools in TROMLAB.

The components of interpreter are the following:

- *Scanner*
  A single text file containing LSL traits, TROM class specifications, subsystem specification, and an initial event list is taken as input to the scanner. Using Flex the scanner performs lexical analysis and identifies the tokens to be used by the parser.
- *Parser*
  This uses Bison to certify the syntactic correctness of the tokens received from the scanner.
- *Syntax analyser*
  Using predefined grammars for TROM and subsystem this module evaluates the syntactic correctness of tokens received by Bison. Any error at this stage will be communicated by Bison to the user and will terminate the execution of the interpreter.
- *Abstract syntax tree generator*
  An abstract syntax tree is generated for each TROM and subsystem input to the interpreter.

- *Semantic analyser*

  This is a C++ program that uses the well-formedness rules of the formalism to perform simple semantic analysis.

- *Error message handler*

  This is part of semantic analyser functionality. Every semantic error detected will be saved in a file until the end of semantic analysis.

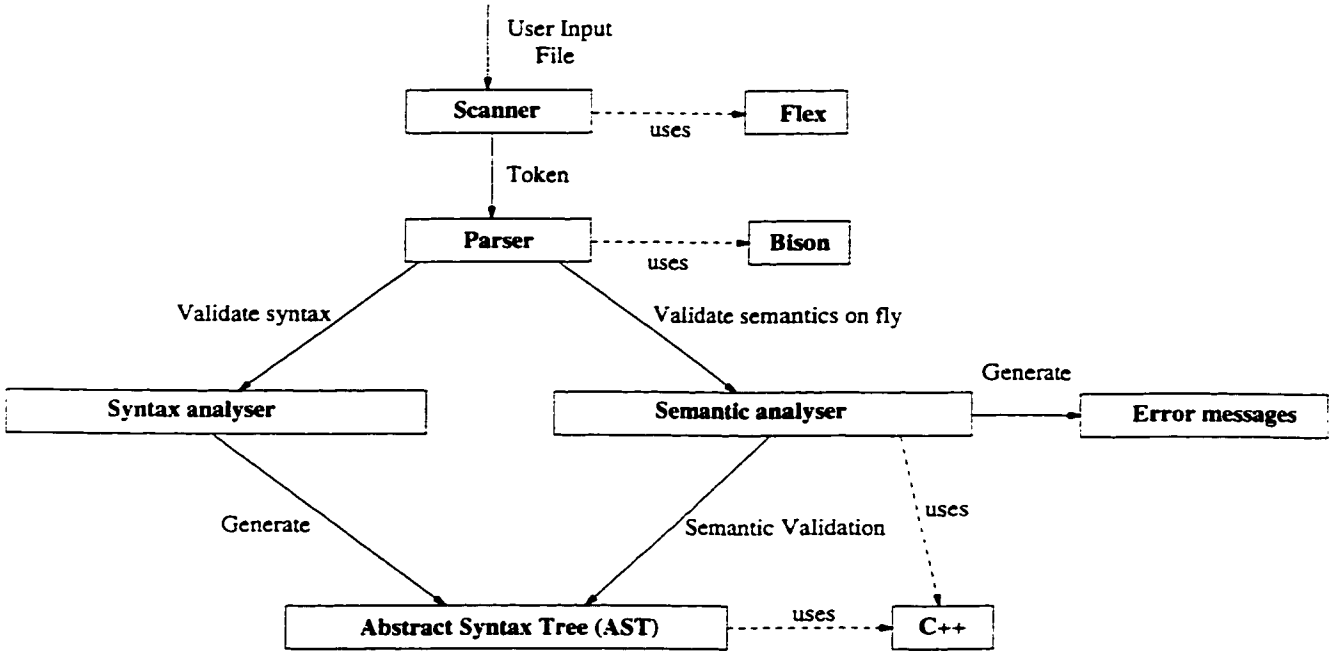


Figure 7: Architecture of interpreter

The interpreter uses YACC and LEX for syntactic analysis and is implemented in C++. This tool had some limitations: all the information had to be in a single file, which makes it difficult to incrementally design a complex system. The data structure generated by this tool also has several limitations. This will be discussed in Chapter 3.

2.3.2 The simulator

The simulator tool was designed and implemented by Muthiayen [Mut96]. This work was started in parallel with the work on the interpreter. The simulator interfaces with the abstract syntax tree built by the interpreter to extract the information for

simulation. It builds a simulation event list to keep track of all outstanding events in the system. The simulator can work in one of two modes:

- *Debugger mode:* In this mode the developer can, at the end of every handled event, invoke the debugger and use it to query the system. The system can be rolled back and new events can be injected.

- *Normal mode:* In this mode the simulation will go on uninterrupted until the system goes into a stable state. The result of a the simulation is one scenario of what could happen, given the initial set of events.

The simulation tool consists of the following components:

- *Simulator*: It consists of an event handler, a reaction window manager, and an event scheduler.

  - Event handler is responsible for handling the events which are due to occur and detects the transition which the event will trigger.

  - The reaction window manager is responsible in activating the computational step to handle the transition causing events to be fired, disabled or enabled.

  - The event scheduler causes an enabled event to occur at a random time within the corresponding reaction window. It schedules output events through the least recently used port using a round robin algorithm.

- *Consistency checker*: It ensures the continuous flow of interactions by detecting deadlock configurations.

- *Validation tool*: It consists of a debugger, a trace analyser, and a query handler.

  - The debugger supports system experimentation by allowing the user to examine the evolution of the status of the system throughout the simulation process. It also supports interactive injection of simulation event, and simulation rollback to a specific point in time.

  - The trace analyser includes facilities for the analysis of the simulation scenario. It gives feedback on the evolution of the status of the objects in the system, and the outcome of the simulation event.

- The query handler allows examining the data in the AST for the TROM class to which the object belongs, and supporting analysis of the static components during simulation.

- *Object model support*: It supports the specification of the TROM classes and the evaluation of the logical assertions included in the transition specifications.

- *Subsystem model support*: It creates subsystems by instantiating included subsystems from object and port links.

- *Time manager*: It maintains the simulation clock updating it regularly. It allows setting the pace of the clock to suit the needs of analysis of simulation scenarios. It also allows freezing the clock while analysing the consequences of a computation.

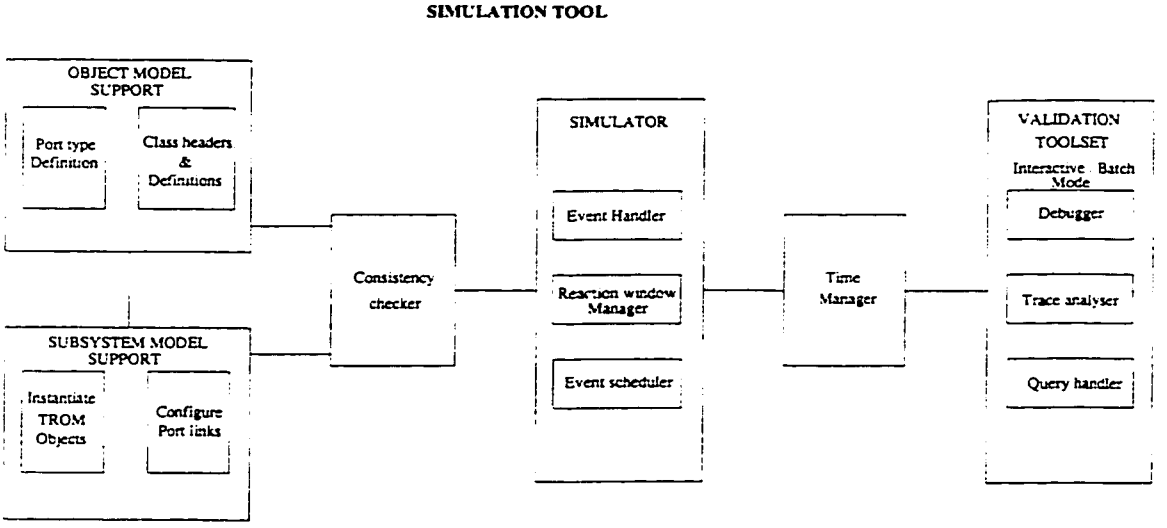


Figure 8: Architecture of simulation tool

Chapter 3

Modification to the Initial Design

Evolving systems need to have a flexible design with abilities to absorb changing requirements with minimal changes to the design. In the absence of a flexible design, it may be necessary to re-engineer and rebuild several of the system components. The initial TROMLAB design is an example of a design which can not be adapted to a graphical user interface front-end. However, GUI has been recognised as an important requirement for the usability of the entire system. Moreover reasoning with partial subsystems was not possible. This is a severe limitation of a large system for which different components may become available at different times. One of the goals of reasoning is to be able to reason modularly and compose the consequences. This is the major reason that a re-engineering of the initial design was undertaken.

3.1 Necessity for re-engineering

The three important criteria for TROMLAB design have been identified as *scalability*, *portability*, and *flexibility*. The TROM methodology provides the support to design systems in a compositional and incremental fashion. This can be translated to the implementation layer only if the language of implementation allows composition and specialisation of class instances. The current implementation in C++ does not meet the above criteria:

1. The *Interpreter* program required one input file containing the textual specifications of all TROMs, subsystems, and LSL descriptions. Moreover, it required

that the data be organised in a certain order. This requirement is quite stringent, acts against the principles of modular and incremental development of the system. Separate compilation of each specification was not possible. Hence, GUI facilities such as individual composition and compilation of TROMs could not be handled by the *Interpreter.*

2. The *AST* constructed by the interpreter had a complex data structure, and its interface to the simulator was poorly designed. Consequently, it became necessary to write interface functions whenever a need arose. This posed severe problems in the maintenance of the *Interpreter.*

3. The language of implementation was C++, which can not be integrated with some of the currently available graphical libraries for Unix platform. In particular, graphical libraries for Unix are written in Java, and do not interface with C++.

4. The current programs run under Unix and are not portable to other platforms.

The revised model of **TROMLAB** environment is shown in Figure 9. It consists of the following components, each designed and implemented to meet the three criteria stated above:

1. *Interpreter:* It should be possible to type check and compile one specification at a time. The order of input is irrelevant. It should be possible to interface with GUI, the simulator, and the verifier. The capabilities of the modified interpreter are discussed in the next section.

2. *Simulator:* It should be possible to simulate any subsystem that has been type checked by the *Interpreter.* It should be possible to view the simulated scenarios and histories through GUI. The capabilities of the modified simulator are discussed in the next section.

3. *Browser:* This tool has been implemented in Java [Nag99]. It can be invoked from within the GUI or it can be invoked as a stand-alone tool. The user can view LSL traits, TROMs, and subsystems from the reuse library database, and query the system for their versions and dependencies.

4. *UMLRT support:* This tool is the front-end for visually composing reactive system specifications using UMLRT support. Class diagrams, state charts, sequence diagrams, and collaboration diagrams can be constructed using Rose. Using stereotypes, an extensional facility in UML, a minimal set of extensions has been provided in to model real-time reactive systems in Rose. The UMLRT support [Oan99] extracts the information from these models and generates formal specifications in the syntax defined in TROM methodology.

5. *Reasoning system:* This tool is the subject of this thesis. It will be described in more details in the following chapters. It gives the user the ability to query the simulated scenario and reason about changes to the past and understand the future consequences due to such changes.

6. *GUI:* The graphical user interface provides a comprehensive interaction facility: it interfaces with Rose/UML tool for composing specifications graphically, which is interfaced with interpreter for syntactic and semantic analysis; simulation scenarios can be viewed, and queries of the *Reasoning System* can be composed, verification steps can be viewed.

7. *Verification Assistant:* The *Simulator* with *Reasoning System* constitute the validation tool. A tool to automatically generate axiomatic descriptions of specifications from the abstract syntax tree is being built now [Pom99]. The results produced by this tool will serve as an input to a mechanised verifier that is being designed.

Integrating all these components in TROMLAB to meet the three design principles cited earlier demand the following:

1. An Object oriented development environment:

2. A good graphics library which supports GUI development:

3. Need for compatibility between different components: UML RT support has been developed under Windows platform; the browser has been implemented in Java.

Based on these constraints we have chosen Java as the language of implementation for the re-engineered components as well as the yet to be implemented components of the verifier.

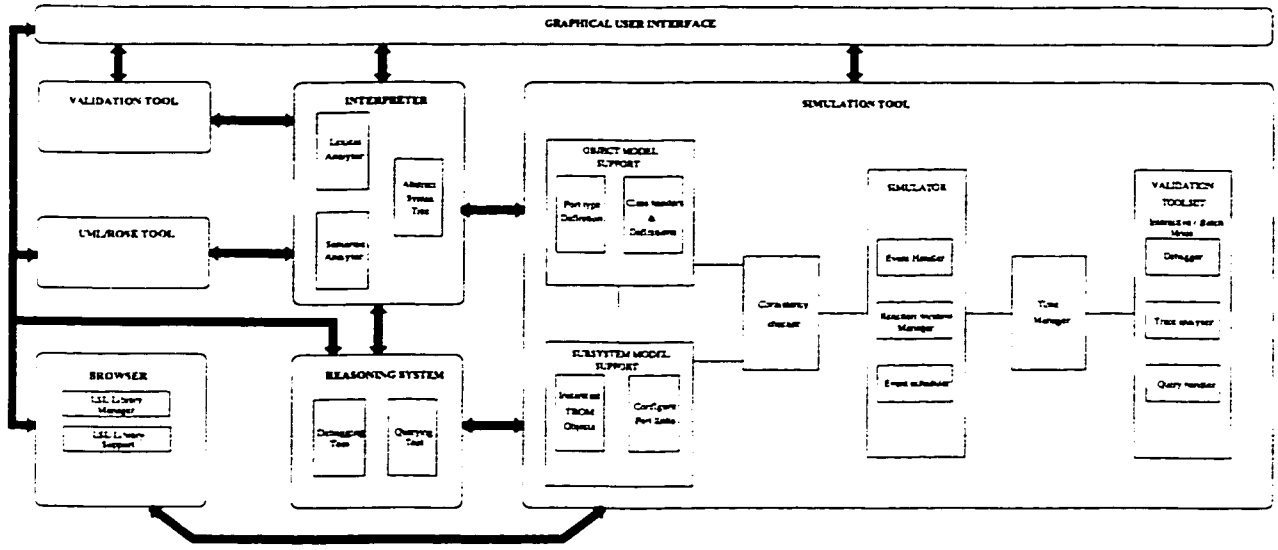


Figure 9: Future TROMLAB environment

3.2 Improvements

It is the *Interpreter* that required a totally new design and implementation. The major changes in the *Animator* include its interfacing to the new *Interpreter*, additional query handling facilities, and enhancements to simulation event list.

3.2.1 Interpreter

1. *Scanners:* Having a single scanner makes the design process harder for the user. The user has to create all the formal specifications at the same time before it can be checked for syntactic correctness. It is quite hard for a single scanner to generate easy-to-understand error messages for a large system consisting of numerous specifications. Whenever a new specification is added to an existing set of specifications it would require re-compilation of the whole set of specifications. A more efficient technique is to have separate scanners, one for each type of component. In the new design we have constructed separate scanners, one for LSI trait, one for TROM class specification, one for SCS, and one for simulation event list. This makes it easier for the user to design, debug and validate different components independently before doing the actual semantic analysis. As a result, the user can reuse the compiled components of any one type without having to wait for the compilation of other specifications. Thus,

the new design conforms to the principle of separation of concerns ingrained in OO methodology and is faithful to the three-tier methodology stated in Chapter 2.

2. *Error messages:* In the old design the error messages were generated by *Flex* and *Bison*. Hence, the messages were neither specific to any one specification nor sufficiently explanatory for the user to understand and correct the errors. In the new design, although *JavaCC* tool is used to parse and compile the specifications, the error messages are not handled by *JavaCC*; instead, the error messages generated by the new *Interpreter* module are quite specific to the source of errors.

3. *Changes to the Grammar:* According to the previous grammar in the *configure* section of the SCS the user could not specify the name of the ports, and in turn it was generated by the *Interpreter* itself based on the cardinality of the specific port type. In the new grammar the user has to specify the port name for each TROM object in the configure section. The other changes to the grammar were made in the initial simulation event list. The name of SCS was added, along with the port type name added to the initial events. This change triggered changes to the semantic analyser. The description of the Grammar is in the Appendix A.

4. *Semantic analysis:* In the previous design the semantic analysis was conducted in two stages: on the fly analysis and AST validation. In the new design, semantic analysis also conforms to the principle of encapsulation in OO technology: semantic analysis internal to a class specification, and semantic analysis relating objects in a subsystem configuration. When a class is syntax checked, it is also semantically validated independently for its encapsulated properties. Once a class is semantically checked and a subsystem of objects is created, the user can initiate the second phase of semantic analysis which does the semantic validation related to the different objects in the subsystem.

5. *AST Structure:* The structure of AST has been simplified. The Figure 10 below describes the new AST structure.

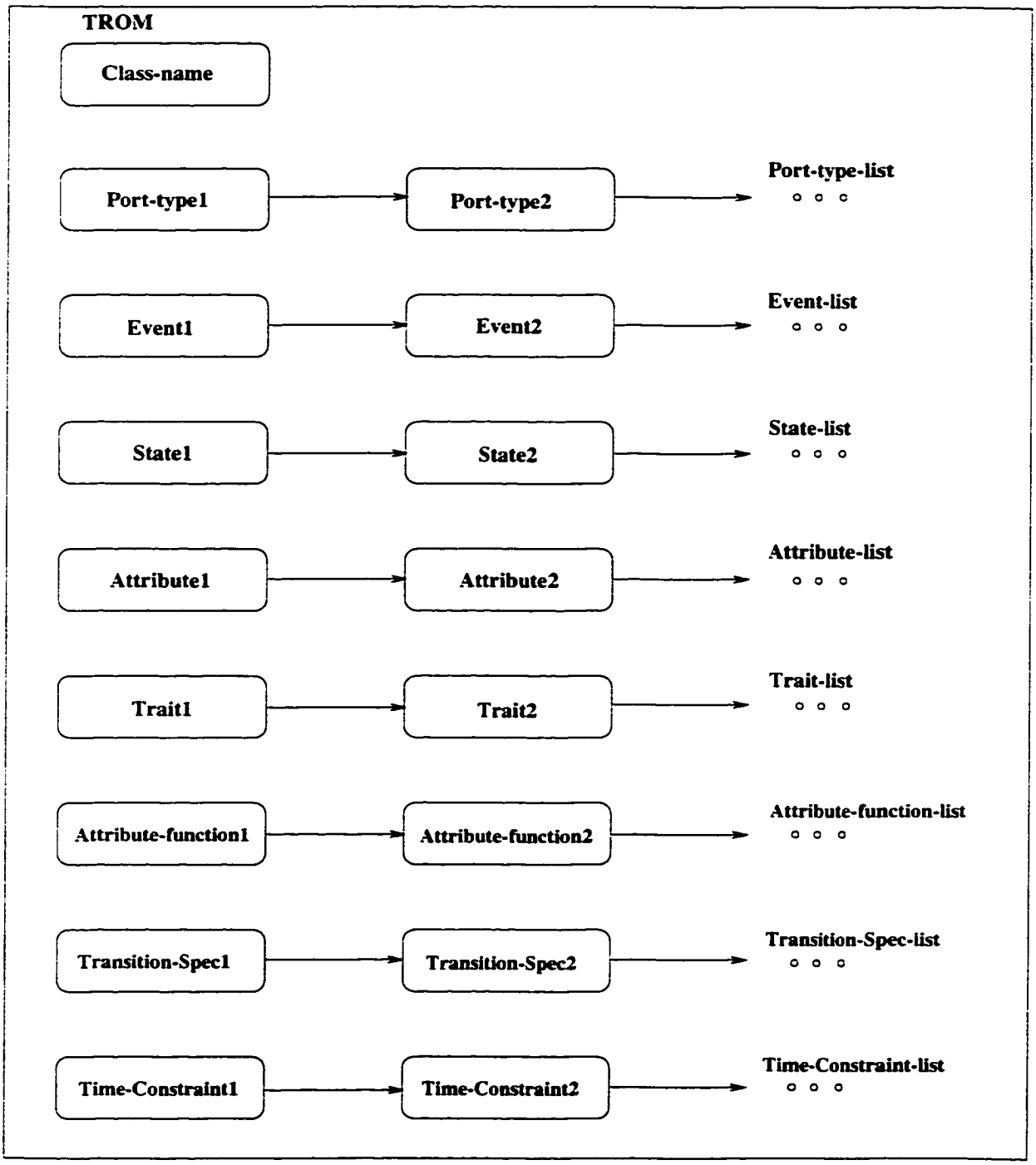


Figure 10: AST Structure

3.2.2 Simulator

1. *Object Model Support:* Due to the changes in the AST structure the existing Object model support needed several modifications. Consequently the way in which the assertions(port, enabling, and post) were evaluated had to be modified.

2. *Simulation Event:* The existing simulation event structure was augmented to have an attribute pointing to the causing event facilitating the tracing of history. This is helpful for later additions, especially in the *Reasoning System.* Consequently, various data structures had to be modified to manipulate the new attribute.

3. *Query Handler:* The simulation tool provides the user with a rollback option. In the previous design the rollback would remove all the events that were scheduled after the time of rollback including the output unconstrained events. Since these events are external to the system, the new design does not remove these events even if they are scheduled after the rollback time. Consequently, these events had to rescheduled.

4. *Event Scheduler:* The simulation tool is capable of handling only deterministic transitions, i.e. only one unconstrained transition going out from a single state. In the case study of *Robotics Assembly* given the later chapter, we encountered a scenario where we had more than one unconstrained transition going out from a single state. In order to solve this non-determinism we had to make few changes to the *EventScheduler* in order that it can handle the non-determinism.

5. *LSL Library Support:* The Simulation tool supports only the *Set* trait. In order to facilitate the design process, we added a few commonly used LSL traits like *Stack*, and *Queue* according to their definitions as part of the *Browser*

Chapter 4

Design and Implementation of the Modified Interpreter and Simulator

In this chapter we compare the new design of *Interpreter*, and the *Simulator* with their old designs to emphasise the significant improvements made according to the description given in the previous chapter. We also discuss the tools which were used to implement the *Interpreter*. The *Reasoning System* is built to work synchronously with the new system.

4.1 Class diagrams

The class diagrams of the old and new design of *Interpreter*, and the *Simulator* are drawn using OMT notation. There are major design changes to the *Interpreter* with regard to the design of the classes, and relationship between the classes. The old design of the *Interpreter* was more rigid, and complex with no scope for further improvements, which motivated us towards doing a more flexible design. We took this opportunity to implement the improvements of the *Interpreter* that are described in the previous chapter. There are only minor design changes for the *Simulator*, i.e. the designs differ in the way the classes are structured, and the relationship between them.

4.1.1 Interpreter

The class diagram for the old *Interpreter* consists of one class *Slink* which is inherited by the classes *Btree node*, *Configure*, *Name-t*, *Att-func*, and *State pair*. The class *Btree node* encapsulates the structure of logical expressions arising in transition specifications. A high-level class diagram of the old *Interpreter* is shown in Figure 11. A detailed class diagram of the old *Interpreter* is shown in Figure 12 and Figure 13.

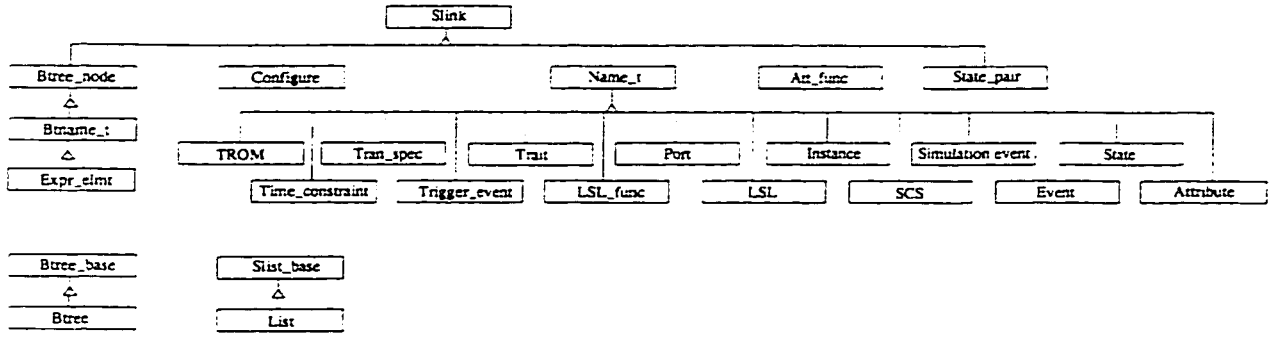


Figure 11: Interpreter Class diagram (Old)

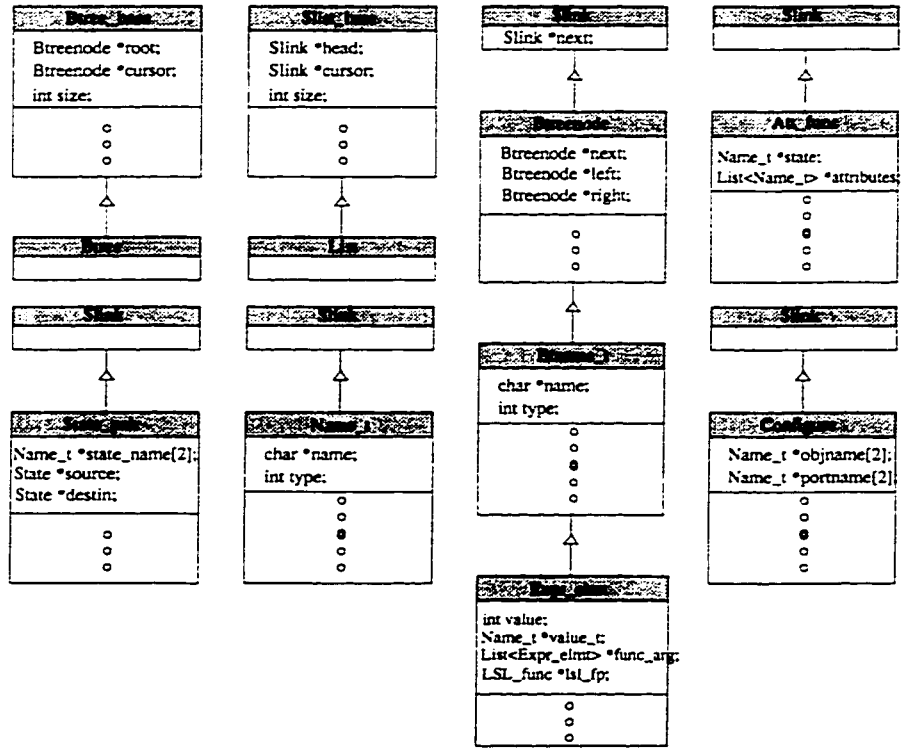


Figure 12: Interpreter Class diagram - Detailed (Old)

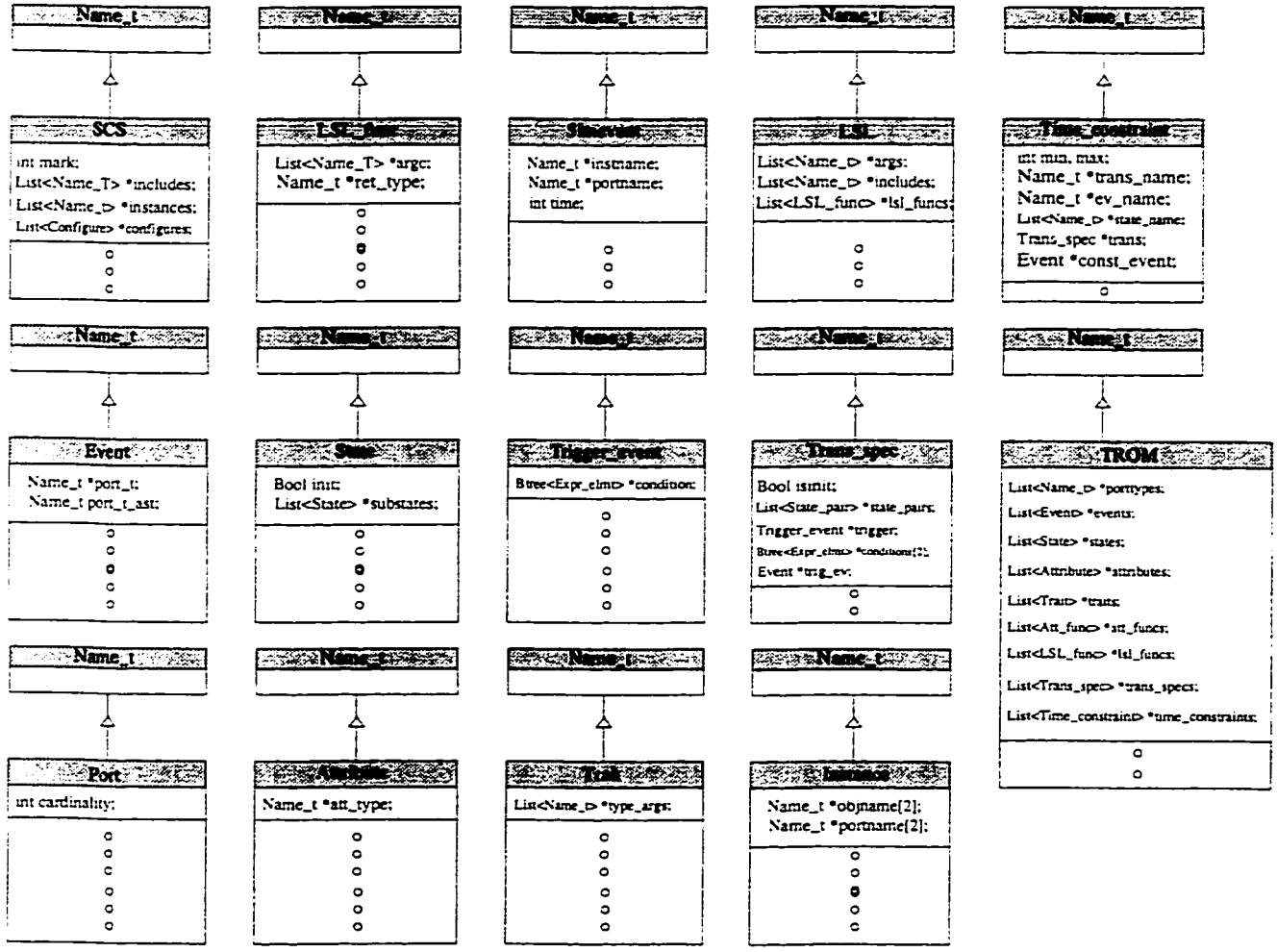


Figure 13: Interpreter Class diagram - Detailed (Old)

The high-level class diagrams of the new *Interpreter* are shown in Figure 14. A detailed class diagram of the new *Interpreter* is shown in Figure 15 and Figure 16.They reflect the true OO features inherent in the problem domain: an abstract syntax tree is an aggregation of *LSL trait*, *TROMclass*, *SCS*, and *SCSSimEv*. These are precisely the classes required to model the entities in the three tires, and the simulation events; the detailed class diagram for each class shows the internal structures and the interface. These diagrams explicitly convey the modularity in the design and the coupling between classes - modifying any one class will not affect any other class.

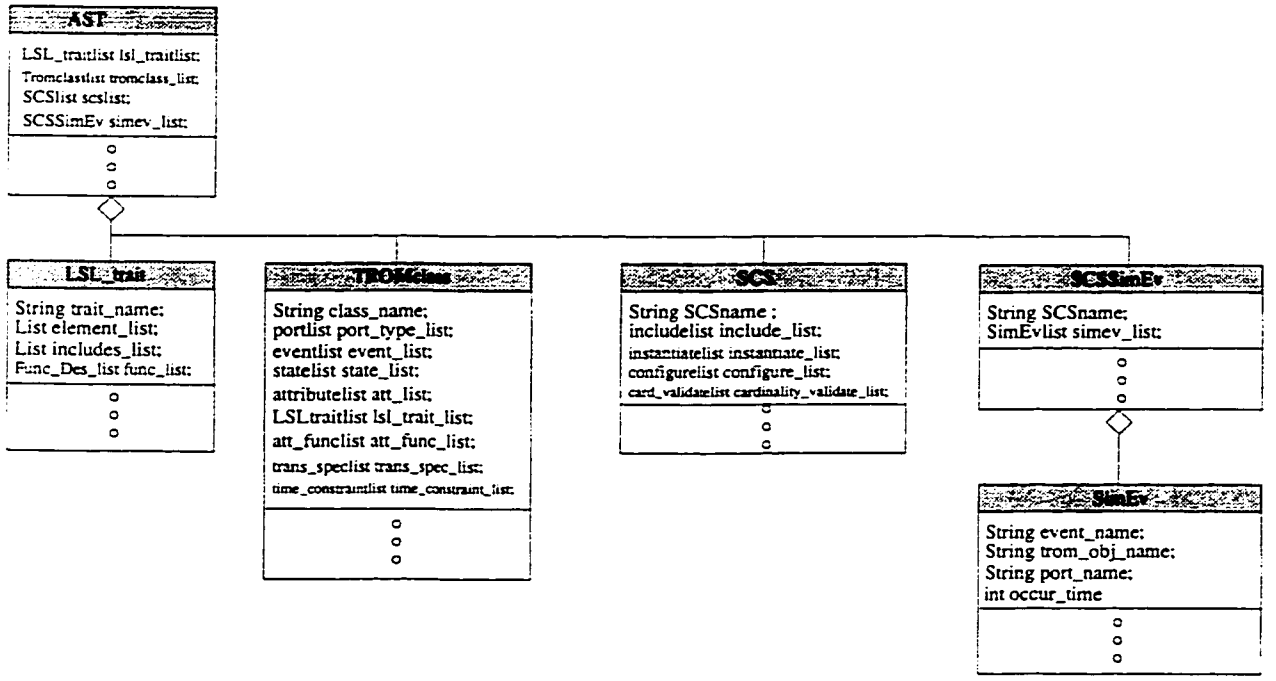


Figure 14: Interpreter Class diagram (New)

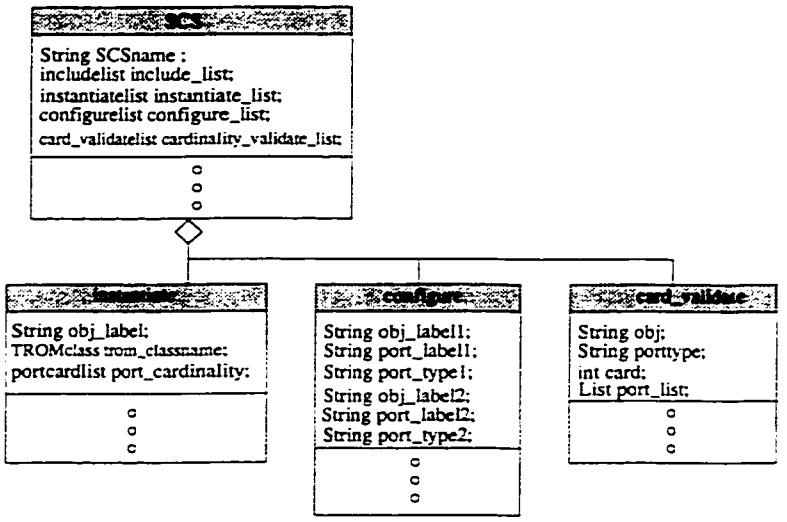


Figure 15: Interpreter Class diagram - SCS (New)

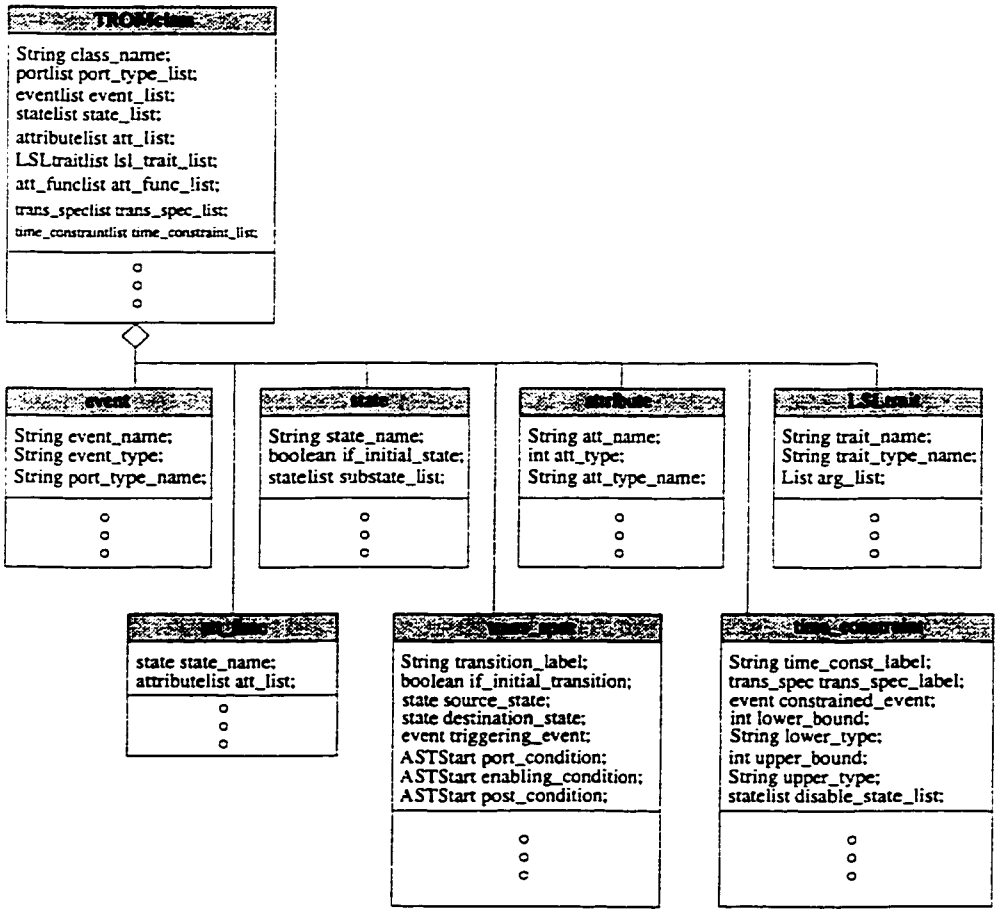


Figure 16: Interpreter Class diagram - TROMclass (New)

4.1.2 Simulator

Class diagram: Since there were no major changes in the design of Simulator, we only show the modified class diagrams. The modifications are based on the improvements suggested in the previous chapter. The detailed class diagrams are shown in Figure 17, Figure 18, and Figure 19.

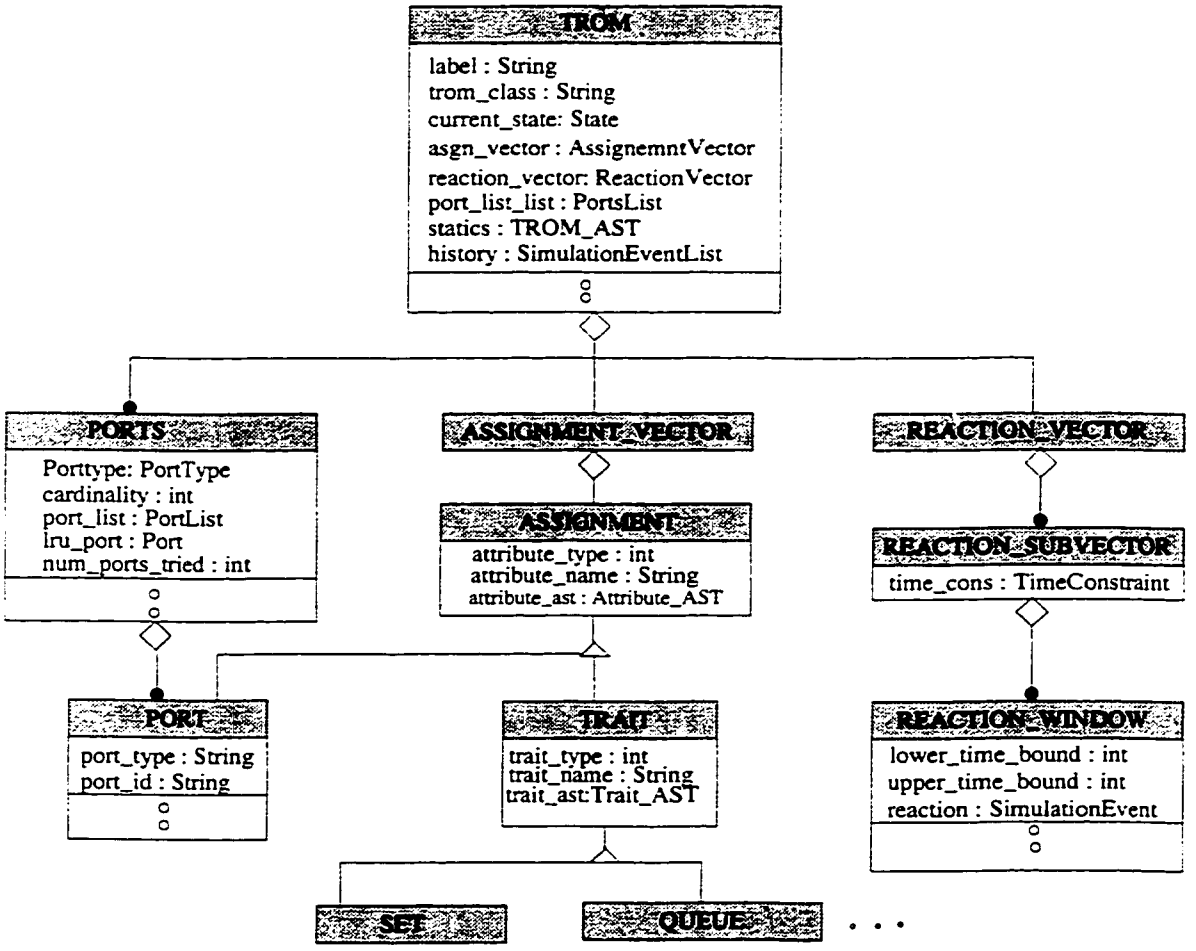


Figure 17: Simulator Class diagram - TROM class diagram(New)

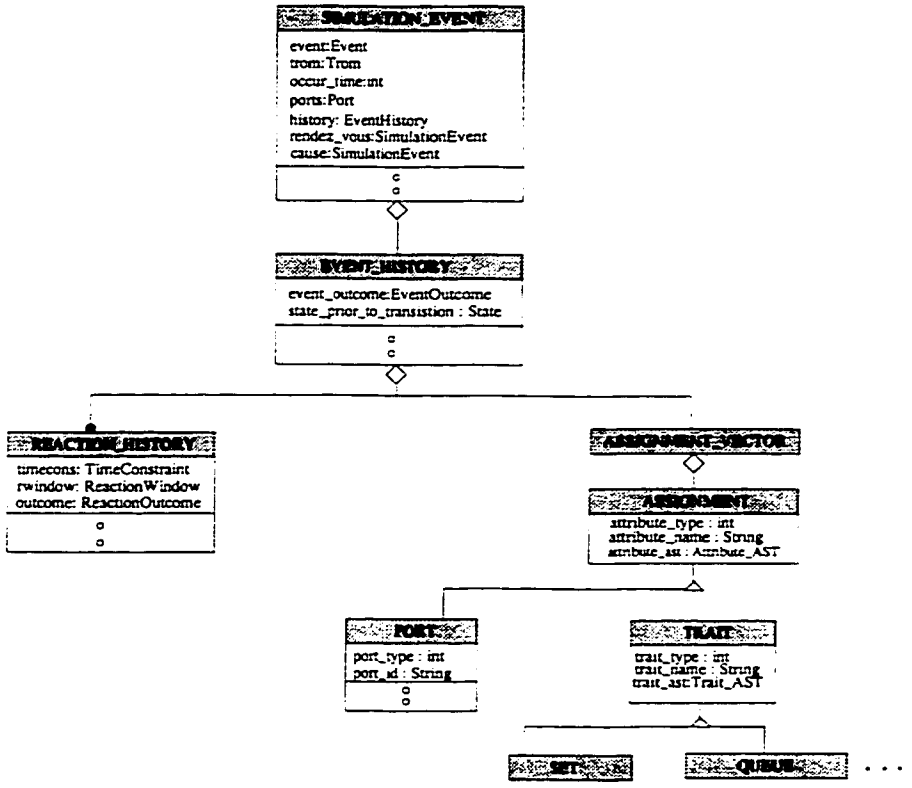


Figure 18: Simulator Class diagram - Simulation Event Object model (New)

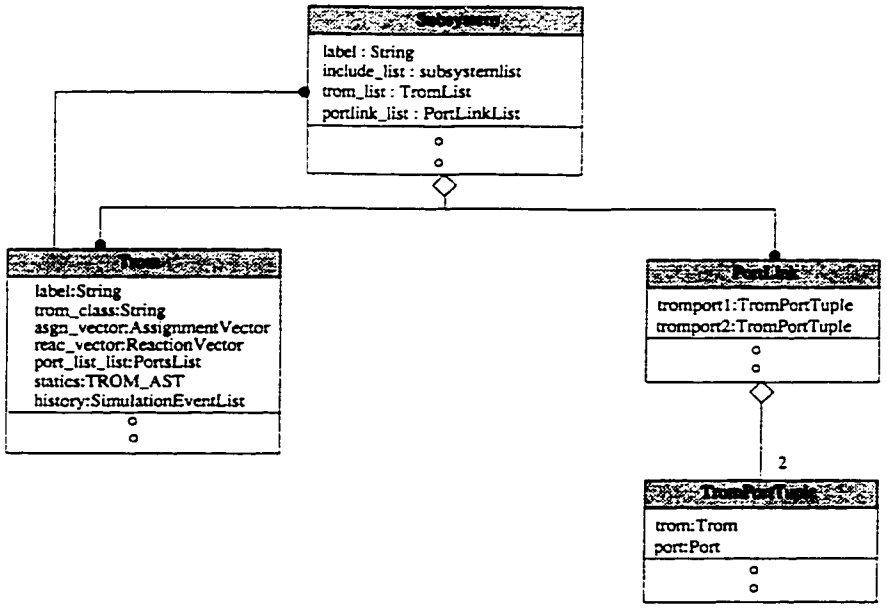


Figure 19: Simulator Class diagram - Subsystem Object model (New)

4.2 Language of choice

We have chosen Java as the language of implementation due to the reasons mentioned in the previous chapter namely

- An Object oriented development environment,
- The need to support portability,
- Good graphical library support.

This choice smoothly integrates the different components of TROMLAB with GUI. We use *JavaCC* and *JJTree*, which are preprocessors for Java, to generate the parser(s) as part of the *Interpreter*.

4.2.1 JavaCC

Java Compiler Compiler (*JavaCC*) is currently the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognise matches to the grammar. In addition to the parser generator itself, *JavaCC* provides other standard capabilities related to parser generation such as tree building, actions, and debugging.

JavaCC is a Java parser generator written in Java. It produces pure Java code. Both *JavaCC* and the parsers generated by *JavaCC* can be run on a variety of Java platforms. JavaCC generates top-down (recursive descent) parsers as opposed to bottom-up parsers generated by other tools, such as *YACC*. This allows the use of more general grammars (although left-recursion is disallowed). Top-down parsers have other advantages (besides allowing more general grammars):

- it is easier to debug,
- the ability to parse to any non-terminal in the grammar, and
- and the ability to pass values (attributes) both up and down the parse tree during parsing.

The lexical specifications such as regular expressions, strings, etc. and the grammar specifications (the BNF) are written together in the same file. It makes the grammars easier to read (since it is possible to use regular expressions inline in the grammar specification) and also easier to maintain.

4.2.2 JJTree

JJTree is a preprocessor for *JavaCC* that inserts parse tree building actions at various places in the *JavaCC* source. The output of *JJTree* is run through *JavaCC* to create the parser. By default, *JJTree* generates code to construct parse tree nodes for each nonterminal in the language. This behaviour can be modified so that some nonterminals do not have nodes generated, or so that a node is generated for a part of a production's expansion. Although *JavaCC* is a top-down parser, *JJTree* constructs the parse tree bottom up. To achieve this it uses a stack where it pushes nodes after they have been created. When it finds a parent for them, it pops the children from the stack and adds them to the parent, and finally pushes the new parent node

4.3 Implementation

We discuss the implementation of the parsers, the syntax for the specifications, and the interfaces to the other components of TROMLAB system.

4.3.1 Interpreter

The parsers, implemented in *JavaCC* and *JJTree*, are used to build the assertion trees. The other classes are implemented in Java. The input to the *Interpreter* is a textual formal specification file(s). The *Interpreter* parses the file and creates the internal representation of the AST (see Figure 10) as a result of syntax checking and on the fly semantic analysis. If the input specification is not syntactically correct, error messages are given, and AST is not created. Once the user has correctly composed the class specifications, and subsystem specification (which may be compiled independently) the overall semantic analysis for the fully specified system is done. Semantic errors at this stage indicate an incorrect or incomplete system specification. When an object which is not a correct instantiation of a correctly compiled class is referred to in the specification of a subsystem, the user might be referring to a class which was not specified (incompleteness) or the user might be incorrectly referring to an existing object (error). When the specifications are syntactically and semantically correct, the user may use the simulator to analyse its behaviour.

A brief description of the implementation of the four parsers is given below:

1. *LSL trait parser:* The LSL trait parser takes a LSL trait file as input and generates the corresponding objects for that file and adds them to the *AST*. In the same LSL trait file more than one LSL trait can be defined, and these LSL traits will be represented by different nodes in the LSL trait's list. If the user submits more than one LSL trait file for the same system, the resulting objects will be in the same list. An example LSL trait file is shown in the Figure 2 in the Chapter 2.

   On the fly semantic checks performed on this file is as follows:

   (a) Trait names should not be duplicated in the *Includes* section.

   (b) A Trait cannot include itself.

   (c) No duplicate functions are allowed in the *Introduce* section (Note: two functions can have same name provided their signatures are different).

   (d) The return type and the parameter types of a function defined in *Introduce* should be defined either in the *Includes* section or in the signature part of the trait. (Note: Integer and Boolean type are assumed to be defined. Int or Integer refers to an integer type, and Bool or Boolean refers to a Boolean type.)

   All these semantic checks are done independently of the other sections in the *AST* and are performed at parse time itself.

2. *TROM class specification parser:* The TROM class specification parser takes a TROM class specification file and generates the corresponding objects for that file, and adds them to the *AST*. In the same TROM class specification file there can be more than one TROM class specification defined, and these classes will be represented by different nodes in the TROM class list. If more than one TROM class specification file for the same system is submitted, the resulting objects will be in the same list. An example of TROM class specification is shown in the Figure 3, Figure 4, and Figure 5 in the Chapter 2.

   The following semantic checks are performed while checking the syntactic correctness of TROM files:

   (a) The port types cannot be duplicated.

(b) The event names cannot be duplicated.

(c) The port types used in the event section should be defined in the port section.

(d) Only the input and output events defined in the event section can have ports associated with them.

(e) There is only one initial state.

(f) The state names cannot be duplicated.

(g) A complex state can have only one entry state which is the initial state for that complex state.

(h) The attribute names cannot be duplicated in the attribute section.

(i) If the attribute is of port type then the port type has to be defined in the port section.

(j) The trait names can not be duplicated in the Trait section.

(k) The port types listed in the signature of the Traits have to be defined in the ports section.

(l) The attributes listed in the signature of the Traits have to be defined in the attribute section.

(m) The state names listed in the attribute-function section should be defined in the state section.

(n) The attribute names listed in the attribute-function section should be defined in the attribute section.

(o) The state names listed in the transition specification section should be defined in the state section.

(p) The transition names cannot be duplicated.

(q) The attribute names listed in the transition specification should be defined in the attribute section.

(r) The event names listed in the transition specification should be defined in the event section.

(s) The time constraint names cannot be duplicated.

(t) The transition names listed in the time constraint should be defined in the transition specification section.

(u) The event names listed in the time constraint should be defined in the event section.

(v) The time interval defined in the time constraint should be valid, i.e. the upper bound should be greater than the lower bound.

(w) The set of states listed in the time constraint should contain only the states that are defined in the states section.

3. *SCS parser:* The SCS parser takes an SCS file as input and generates the corresponding objects for that file, and adds them to the AST. In the same SCS file there can be more than one SCS defined, and these SCS will be represented by different nodes in the SCS list. If more than one SCS file for the same system is submitted, the resulting objects will be in the same list. An example of SCS is shown in the Figure 6 in the Chapter 2.

   The following semantic checks are performed while syntax checking an SCS file:

   (a) SCS names listed in the Includes section cannot be duplicated.

   (b) TROM objects defined in the instantiate list cannot be duplicated.

   (c) All the port types listed in the configure section should be instantiated in the instantiate section of this or any of the included subsystem.

4. *Initial Simulation event list Parser:* The simulation event list parser accepts a simulation event list file as input and generates the corresponding objects for that file, and adds them to the AST. An example of simulation event list is as follows:

```
SEL: TCG
   Near, t1, @C1, 3;
   Near, t2, @C2, 5;
   Near, t3, @C1, 7;
end
```

Figure 20: Simulation event list

Since the objects added to the AST have been generated independently, and are however dependent on each other, an overall semantic analysis has to be performed once the user is finished with the design. The overall semantic analysis checks for the following properties:

- Between LSL traits and TROM class specification the following dependencies must hold:

  1. Every LSL trait used in a TROM class has to be defined.
  2. The signature of every LSL trait function used in the assertion expressions of the transition specification section of a TROM class should match the signature defined in the corresponding LSL trait.
  3. The return type of the LSL trait function used in the assertion expression of the transition specification of a TROM class should match the operands used in the expression.

- Between TROM class specification and SCS the following properties should hold:

  1. Every TROM object defined in the *Instantiate* section of a SCS must be an instance of a TROM class in the *AST*.
  2. Every TROM object defined in the *Instantiate* section of a SCS should have its ports associated to the port type defined in the TROM class.
  3. Links can exist between two instantiated TROM objects, or between an instantiated TROM object and an open port of a subsystem included in SCS.
  4. Every subsystem listed in the *Includes* section must have been compiled earlier.

- Between SCS and SCS the following properties hold:

  1. The number of ports of a port type used for a TROM object should be less than or equal to the cardinality of that port type defined in the instantiate section. This has to be checked taking into consideration all the included subsystems in the Include section of SCS.

2. All the TROM objects listed in the configure section should be defined in the instantiate section. This has to be checked taking into consideration all the included subsystems in the Include section of SCS.
3. Port names of the same port type defined in the configure section cannot be duplicated. This has to be checked taking into consideration all the included subsystems in the Include section of SCS.
4. All the TROM objects defined in all the included SCS's cannot have duplicate names.
5. Only compatible ports can be linked.

- Between Simulation event list, SCS, and TROM class specification the following properties hold:

1. Every TROM object listed in the simulation event list should be defined in the SCS or in any one of the included SCS's of that SCS. (The name of the SCS appears in the Simulation event list)
2. For every TROM object listed in the simulation event list, the corresponding event name should be defined in the corresponding TROM class in the event section and this event should be of the type output and unconstrained.
3. For every TROM object listed in the simulation event list, the port name listed should be defined in the SCS or in any of the included SCS's of that SCS for that corresponding TROM object.

4.3.2 Simulator

The simulator was implemented based on the existing design using Java. Simulator makes use of the *AST* generated by the *Interpreter* and generates one of the possible scenarios for the given system and the initial simulation event list.

The simulation steps are as follows [Mut96]:

1. Instantiate TROM objects: Adds the dynamic information(assignment vector, and reaction vector) for each TROM object instantiated in the SCS to be simulated.

2. Instantiate simulation event list: Schedules unconstrained internal events from initial states. Schedules the initial simulation event and their corresponding rendezvous.

3. Handle the events: Traverses the simulation event list with respect to time and handles the events by evaluating the port, pre and post conditions and taking an action accordingly of firing or disabling the corresponding transition, and scheduling the resulting events.

4. Handle the history: Saves the state, assignment, and reaction vector prior to the transition.

5. If the system is in debugger mode, it asks the user after handling of each event if he wants to invoke the debugger.

6. Debugger: The debugger allows the user to perform different kind of queries and also allows to invoke the trace analyser.

7. Trace analyser: Trace analyser allows the user to query the static information of the different TROM objects in the subsystem and of the subsystem itself.

4.3.3 Interfacing with the simulator

Since the design of *Interpreter* was changed drastically from the previous version, there was a major change in the way *Simulator* interfaced with the *Interpreter*. We had to make sure that the *Simulator* could interface with the *Interpreter* to perform its task. Thus in the *Interpreter* we had to implement all the methods used to by the *Simulator*. We had to modify the simulator in certain aspects:

1. In the previous implementation of the *Simulator* the port names were generated automatically, but in the new version the port names are taken from the user in the *Configure* section. Thus the *Simulator* has to interface with the *Interpreter* in order to get this port name list.

2. Since the structure of assertion tree was changed in the *Interpreter*, the evaluation of these assertion tree in the *Simulator* had to be modified. Thus it lead to major modification in the Object Model support.

Chapter 5

Reasoning System: Requirements

Real time reactive systems are very complex. Any error in the design would lead to catastrophic consequences. We need to be able to debug and verify the design before the implementation. The *Simulator* is a very powerful tool, that helps the user to simulate his design. However this tool has some limitation. In simulating complex systems the history of the simulation becomes very big and very hard to manage and understand. The history of the simulation is seen only from one point of view, that is the *Simulation Event List* point of view. It does not allow the user to look at the history of particular events and particular TROM objects. The tool doesn't allow the user to modify the timing of events. This modification of the timing of the events may help the user in seeing different scenarios in a more controlled environment, not relying on the randomness of the simulation. It offers no tools to study the routing of static data structure and the study of possible timing conflicts.
The *Reasoning System* is a good compliment to the simulation tool. The simulation goes forward in time whereas a debugging tool that reasons about the behaviour of the system needs to go backward in time. The *Simulator* that we have described in the previous chapters goes forward in time while keeping a trace of the history. Our *Reasoning System* has three main roles:

- Debugging tool: This role is served by answering questions that give different points of views on the results of the simulation. These questions will allow the user to view the results of the simulation from the TROM object point of view, from the event point of view and from the *Simulation Event List* point of view. They will also allow the user to have a better understanding of what caused

certain events and transition to occur. The queries that will permit this are described the following sections of this chapter.

- Hypothetical queries: This set of queries will allow the user to have more control on the timing of the events. A detailed description of the hypothetical queries is given in the following sections of this chapter.

- Validity of the Specifications: The *Reasoning System* has to give the user a way to see the reachability of the states, and the correctness of the timing constraints. To do that the *Reasoning System* will provide a set of queries that will be described later in this chapter.

5.1 Reasoning System as a Debugging Tool

By answering the following set of queries we will give a clearer image of the history, from different points of views. These queries will help the user in his debugging process. These queries are :

- *Why:* This will tell the user why the system went from one state to another. This query will be invoked when the simulation is stopped in debugging mode. It will tell the user what are the events that caused the system to go from one state to the other. It will also give the reason behind the occurrence of these events. This query gives the user a perspective on the history from the point of view of states.

- *When:* This will give the user an easy way to check for the timing of certain events. It will also allow the user to see when a set of TROM objects were in a certain state, which would be almost impossible if we relied only on the Simulation Event List, given the large number of TROM objects and the complexity of the reactive systems. This set of queries will give the user a perspective on the simulation from the timing point of view. This query will be divided into six different sub-queries:

  1. *When was the system or the specified TROM object in a specific state?:* This query will be invoked when the simulation is stopped in debugging

mode. The user will be able to get a list of time intervals during which the system or the TROM object was in a specific state.

2. *When did the system or a specified TROM object go out of a specific state?:* This query will be invoked when the simulation is stopped in debugging mode. This will allow the user to know at what times did his system or the TROM object gets out of a critical state.
3. *When an event was fired?:* This query will be invoked when the simulation is stopped in debugging mode. It will provide the user with a set of times when a particular event was fired.
4. *When an event was disabled?:* This query will be invoked when the simulation is stopped in debugging mode. It will provide the user with a set of times when a particular event was disabled. That is when the system went into a disabling state.
5. *When an event was enabled?:* This query will be invoked when the simulation is stopped in debugging mode. It will provide the user with a set of times when a particular event was enabled. That is the time when the event that caused this event to be enabled was fired.
6. *When an event was scheduled?:* This query will be invoked when the simulation is stopped in debugging mode. It will provide the user with a set of times when a particular event was scheduled to be fired or disabled later.

- *Show the assignment vector at a particular time:* This query will allow the user to see the values of the different attributes at specific times. Since the assignment vector changes dynamically the *Simulator* does not keep an image of this vector at all times. It would be almost impossible for the user to reconstruct an image of this vector relying on the *Simulation Event List.* This query allows the user to understand better the values of the attributes at certain critical times in the simulation.

- *Show the reaction vector at a particular time:* This query will allow the user to view the outstanding reactions of the reaction vector at a particular time. Since this vector could grow to be a very big data structure as time progresses, the simulator only keeps the necessary information, and the user would not be able to reconstruct this vector relying only on the *Simulation Event List.* This

query allows the user to understand better the outstanding reactions at certain critical times in the simulation.

- *Does the system go into a specific state?:* This will give the user the possibility to see if the system, or a set of objects in the system went into a state during a time interval. This would be very hard without this tool given the large number of TROM objects in a complex reactive system. This query allows the user to debug his design by showing that some objects were in some states when they were not supposed to.

- *Does the system go into a specific state more than once?:* If the user wants to see if there is a pattern in the behaviour of the system this query would allow him to see that behaviour on the system or on a set of objects in the System.

- *Show the TROM status during a time interval:* This query will allow the user to see the variation in a TROM object without taking into consideration other TROM objects in the system It allows the user to isolate and have a better understanding of the behaviour of a particular TROM object.

- *Show the simulation event list of a particular TROM object:* This query will allow the user to isolate the simulation event list of a particular TROM object from the very complex *Simulation Event List* of the system.

5.2 Reasoning Based on Hypothetical Queries

By answering the following set of queries we will give the user possible scenarios to see when the timing of certain events are modified. We will give the user the possibility to insert new events and see the effect they will have on the behaviour of the system. We will also allow the user to remove certain events and understand the effect that it has on the results of the simulation. Theses queries have to respect certain criteria that would prevent them from violating the requirement of the simulation. Theses criteria will be described in more details in Chapter 6.

- *What if we insert an event?:* This query will allow the user to interactively insert new events, thus allowing the study of new scenarios without having to modify the *Original Simulation Event List*, compiling it again and running the

Simulator again from the beginning. Since the user represents the environment, the user is only allowed to insert environmental events, that is output unconstrained events.

- *What if we remove an event?:* This query will allow the user to interactively remove events without having to do this modification on the *Original Event List*, compiling it again and running the *Simulator* from the beginning. Since the user represents the environment the user is only allowed to remove environmental events that is output unconstrained events.

- *What if we reschedule an event?:* This query will give the user more control over the timing of the events and thus modifying the simulation results. This is impossible in the *Simulator* since it schedules the events at random times within the allowable timing interval. The user can reschedule constrained events within the allowable timing intervals. The user can reschedule output unconstrained events to any time since the user represents the environment. The user is not allowed to reschedule input events: in order to do that the user has to reschedule the corresponding output events.

5.3 Using the *Reasoning System* for Validation of the Specifications

There are four types of errors that we attempt to detect in this section namely:

- Is a state reachable inside a TROM object?
- Are there any undesirable routes from one state to the other within a TROM object?
- Is a state reachable inside the SCS?
- Is the route to reach the state in the SCS consistent with timing constraints?

By answering the following queries we will allow the user to detect the above mentioned errors, if any, in the design.

- *Find all the routes between any two states of a TROM object:* Analysing the response the user can determine whether undesirable routes exist in the design. If applied to every state in the TROM object it will allow the user to make sure that all the states in the TROM object are reachable. This is very useful since every state in the TROM object should be reachable. A state may not be reached only if there is an error in the design of the TROM object. Figure 21 shows this kind of error.

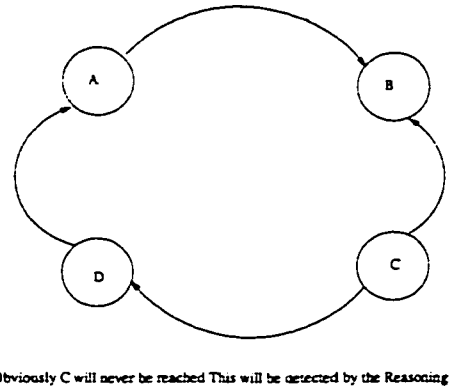


Figure 21: Unreachable state within a TROM

- *Find a route to a specific state of a TROM:* This will allow the user to find a set of events that will lead the object to go from its initial state to the specified state, *taking into consideration its interaction with all the related TROM objects in the system.* Every state in every TROM object should be reachable. It may be the case where a state can only be reached if an input event occurs. If this input event is supposed to come from a TROM object that is not connected via the SCS, we will never reach this state. Figure 22 shows how this kind of errors may occur. If TROM object A and TROM object B are the only two objects in the SCS, since the output event in TROM object B is not equivalent to the input event I in TROM object A, the state B of TROM object A will never be reached. The needed input event will not occur at any time. This will be detected by the Reasoning System.

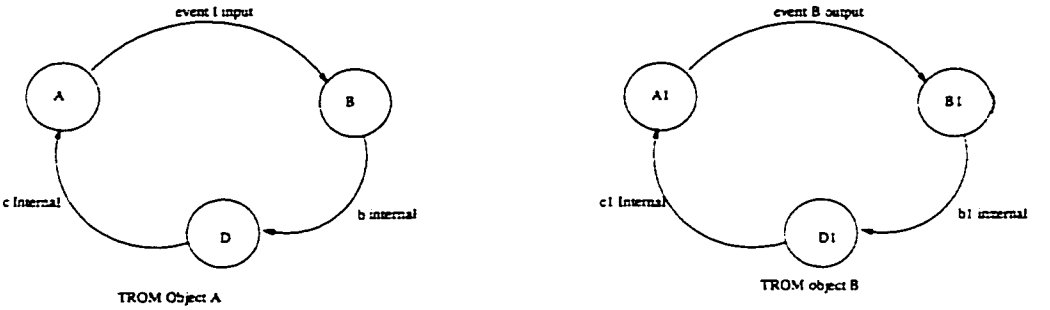


Figure 22: Unreachable state in SCS

This query will also check if the route it found does not have any timing inconsistencies due to timing constraints.

Chapter 6

Reasoning System : Design and Implementation

When designing the *Reasoning System* we had a choice between designing it as an independent module, or as a module that is attached to the existing components, namely the *Simulator* and the *Interpreter*.
The *Reasoning System* has to work with the *Simulator* and with the *AST* in a very tightly coupled manner. This *Reasoning System* is designed to work in the TROMLAB environment. It needs the static data generated by the *Interpreter* and the dynamic data generated by the *Simulator*. It is not designed to work in any other environment. Due to this continuous need of interaction with the existing components of the TTROMLAB environment we decided to design the *Reasoning System* as a module that is attached to the existing components.
When we started with the re-engineering process described in the previous chapter, we decided to use this opportunity to modify the *Simulator* and the *Interpreter* by adding new methods and data members in different classes, so that we will use these methods later on by the *Reasoning System*.
The class diagram of the *Reasoning System* is shown in Figure 23.

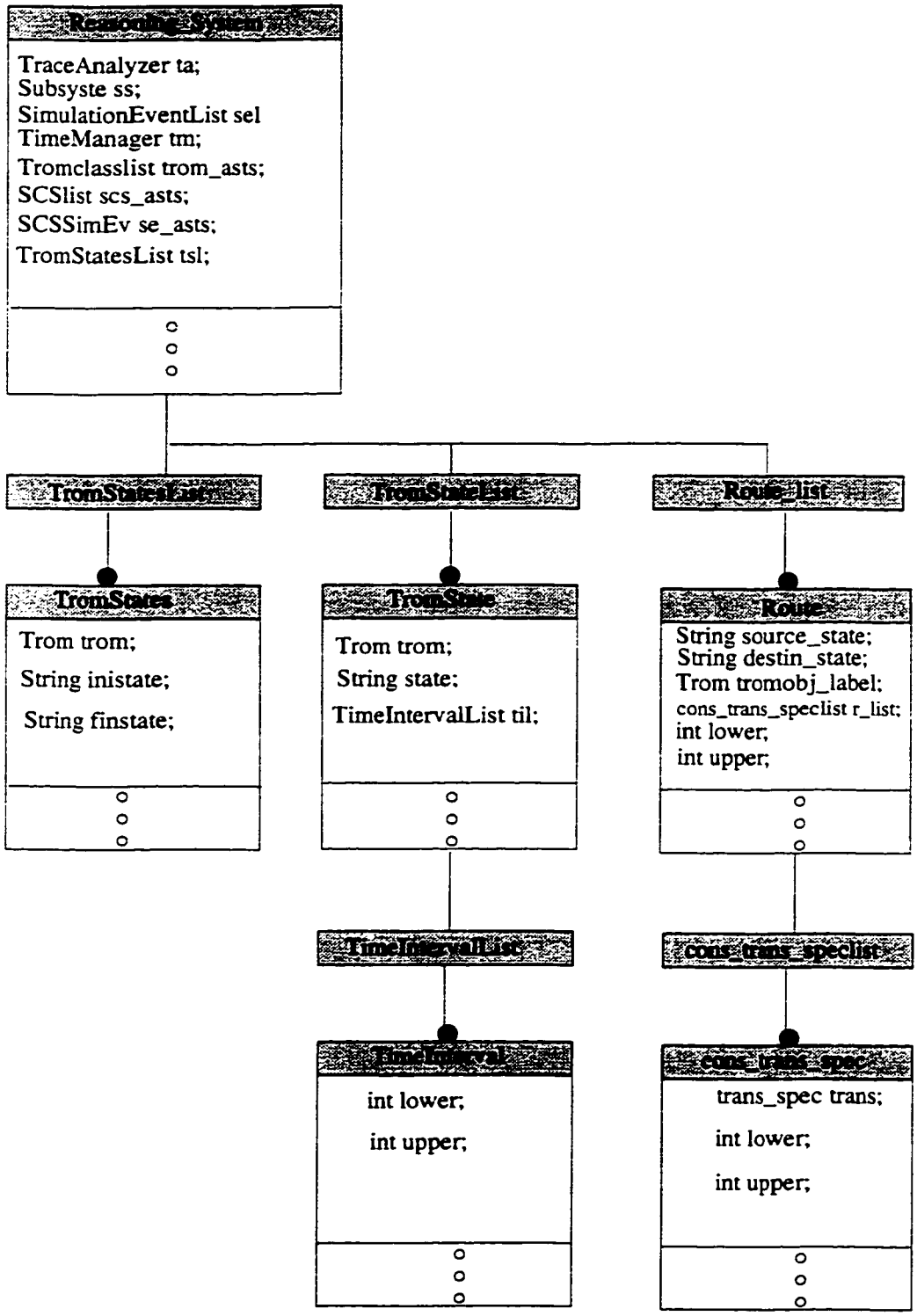


Figure 23: Class Diagram Of the *Reasoning Systemm*

We divide the types of queries in the *Reasoning System* into three categories:

- Debugging tool.
- Hypothetical queries.
- Validation of the Specifications.

We discuss each query type in each section below. In each section we will have a Use Case Analysis, and we will describe the precondition and the post condition of every query. We will also describe some of the algorithms used to implement these queries.

6.1 Debugging tool

All the queries in this section do not affect the results of the simulation. They are related to the history and help the user have a better understanding of the results of the simulation by giving the user different points of views on the results obtained. All these queries will be invoked when the simulation is stopped in debugging mode. The *Reasoning System*, acting as a debugger, will scan the history of the simulation in different ways allowing the user to get a clear image of this history.

Use Case Analysis

The Use Case Diagram illustrated in Figure 24 contains three actors: the user, the *Reasoning System* and the *Simulator*. Once the simulation is stopped in debugging mode the user invokes a query in the *Reasoning System*. The *Reasoning System* then uses the methods that we created to scan the history an returns the result to the user. The user then invokes the *Simulator* to either continue with the simulation or stop it.

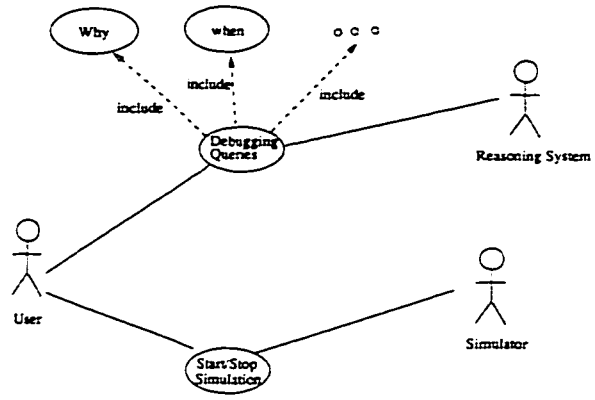


Figure 24: Use Case Diagram For Debugging Tool

Debugging Queries

We describe each query by showing the required input, the precondition and the expected output. In some cases we will describe the algorithm used to achieve the required results. The queries are the following:

- *Why query*: This query will tell the user why did the system or a part of the system go from state S1 to state S2.
  Input from the user
  The user will enter a list of TROM objects. for each TROM object the user will provide the initial state and the destination state.
  Precondition:
  The entered TROM objects have to be valid TROM objects instantiated in the SCS. The corresponding States have to be valid states.
  Expected output
  For each TROM object entered by the user the query will provide the list of events that led the TROM object to go from state S1 to state S2. The query will also provide the reason behind the occurrence of these events. There are five possible reasons:

  1. The event is an output unconstrained event; that means, it is an environmental event. Theses events are out of the control of the system, they are the stimulus of the environment to the system. The only way that these events can occur is if they are entered by the user in the original simulation event list or entered later by the user using the *Hypothetical Queries* of the

Reasoning System. The reason for these events to occur is that *they were entered by the user.*

2. The event is an output constrained event. The cause would be the event that caused the transition constraining this event.
3. The event is an internal unconstrained event. The cause is the event that led the system to the state where this event was bound to happen.
4. The event is an internal constrained event. The cause would be the event that caused the transition constraining this event.
5. The event is an input event. The cause would be the corresponding output event that synchronised this event and led to it being scheduled.

Algorithm
For Each TROM object entered by the user
Repeat until the end of the Simulation Event List
Scan the Simulation Event List *to determine when was the TROM in the initial state entered by the user.*
From that point scan the Simulation event List *to determine when the TROM is in the destination state.*
display the Simulation Events related to this TROM between the two times.

- *When Query*: this query is divided into six sub-queries namely:
  - *When was the system in a given state?* This query will show the user time intervals during which the system or a part of the system was in a particular state.
    Input from the user
    The user will enter a list of TROM objects. For each TROM object the user will provide a state.
    Precondition:
    The entered TROM objects have to be valid TROM objects instantiated in the SCS. The corresponding States have to be valid states.
    Expected output
    For each TROM object entered by the user the query will provide the

list of time intervals during which the corresponding TROM object was in the required state, and the query will provide the intersection of these timing intervals, representing the time interval when the system was in the required state.

Algorithm

For Each TROM object entered by the user

Repeat until the end of the Simulation Event List

Scan the Simulation Event List *to determine when was the TROM in the state entered by the user.*

display the timing intervals.

Display the intersection of the timing intervals obtained.

- *When did the system go out of a given state?* This query will show the user the times at which the system or a part of the system went out of particular state.

  Input from the user

  The user will enter a list of TROM objects. for each TROM object the user will provide a state.

  Precondition:

  The entered TROM objects have to be valid TROM objects instantiated in the SCS. The corresponding States have to be valid states.

  Expected output

  For each TROM object entered by the user the query will provide the list of time intervals during which the corresponding TROM object was in the required state, and the query will provide the intersection of these timing intervals, representing the times when the system went out of the required state.

- *When was an event fired?* This query will tell the user the times at which a particular event was fired.

  Input from the user

  The user will enter an event. The user can also provide a list of TROM objects. If the user does not enter this list of TROMs then the query will answer for the entire system.

  Precondition:

The entered event has to be a valid event in the TROMs entered.

Expected output

The query will provide a list of times when the specified event was fired.

- *When was an event disabled?* This query will tell the user the times at which a particular event was disabled.(that is the time at which did the system go into a disabling state)

  Input from the user

  The user will enter an event. The user can also provide a list of TROM objects. If the user does not enter this list of TROMs then the query will answer for the entire system.

  Precondition:

  The entered event has to be a valid event in the TROMs entered.

  Expected output

  The query will provide a list of times when the specified event was disabled.

- *When was an event enabled?* This query will tell the user the times at which a particular event was enabled.(That is when did transition causing this event to be enabled was fired.)

  Input from the user

  The user will enter an event. The user can also provide a list of TROM objects. If the user does not enter this list of TROMs then the query will answer for the entire system.

  Precondition:

  The entered event has to be a valid event in the TROMs entered.

  Expected output

  The query will provide a list of times when the specified event was enabled.

- *When was an event scheduled?* This query will tell the user the times at which a particular event was scheduled.(That is at which time was this event scheduled to be fired or disabled)

  Input from the user

  The user will enter an event. The user can also provide a list of TROM objects. If the user does not enter this list of TROMs then the query will answer for the entire system.

  Precondition:

The entered event has to be a valid event in the TROMs entered.
Expected output
The query will provide a list of times when the specified event was scheduled.

- *Show Assignment Vector at Given Time*: This query will give the user the status of the assignment vector at a particular time.
  Input from the user
  The user will enter a time.
  Precondition:
  The entered time has to be less then the current simulation time.
  Expected output
  For each TROM object the query will provide the value of the attributes at that given time.

- *Show Reaction Vector at Given Time*: This query will give the user the outstanding reactions of the reaction vector at a particular time.
  Input from the user
  The user will enter a time.
  Precondition:
  The entered time has to be less then the current simulation time.
  Expected output
  For each TROM object the query will provide the value of reaction vector at that given time.(That is the outstanding transitions at that time.)

- *Reachability*: This query will tell the user if the system went into a specific state during the simulation. This query is a different way of asking the query "*When was the system in a specific state?*". If the time interval intersection has at least one element the answer will be yes.

- *Multiple Entry*: This query will tell the user if the system went into a specific state more then once during the simulation. This query is a different way of asking the query "*When was the system in a specific state?*". If the time interval intersection has at least two element the answer will be yes.

- *Show TROM Status During Time Interval:*
  This query will give the user an image of status of a TROM at every time during the entered time interval. That is at which state is the TROM object and what are the values of the attributes at that time.
  Input from the user
  The user will enter a TROM object and a timing interval.
  Precondition:
  The entered upper bound of the timing interval has to be less then the current simulation time. The TROM object has to be a valid TROM object.
  Expected output
  For discrete time the query will provide the state of the TROM object, and the attribute values.

- *Show Simulation Event List of a particular TROM object:*
  This query will give the user the *Simulation Event List* of a particular TROM object from the beginning of the simulation until the current time.
  Input from the user
  The user will enter a TROM object.
  Precondition:
  The TROM object has to be a valid TROM object.
  Expected output
  A *Simulation Event List* for that particular TROM object.

6.2 Hypothetical Queries.

All the queries in this section change the results of the simulation by changing the timing of certain events. All these queries will be invoked when the simulation is stopped in debugging mode, they will permit the user to modify the timing of events, and then run the simulation.

Use Case Analysis

This Use Case Diagram illustrated in Figure 25 contains three actors: the user, the *Reasoning System* and the *Simulator.* Once the simulation is stopped in debugging mode the user invokes a query in the *Reasoning System.* The *Reasoning System* then

uses the methods that we created modify the timing of events. The *Reasoning System* then invokes the *Simulator* to continue with the simulation from an appropriate time.

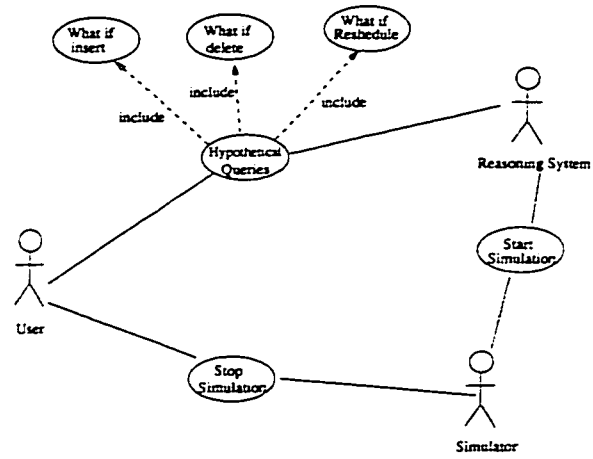


Figure 25: Use Case Diagram For Hypothetical Queries

What If Queries

In this section we will describe each query by showing the required input, the precondition and the expected output. In will describe the algorithm of the *Rollback* which is used by the three queries. The queries are the following:

- *What if we remove an event?*:
  This query is posed when the user want to analyse the consequences of removing an event.
  Input from the user
  The user will enter an event, the TROM object on which this event occurred, the port on which this event occurred and the time this event occurred. We need the time because the same event can occur more then once.
  Precondition:
  The entered event has to be an environmental event i.e. an output unconstrained event. The event time, TROM object and port have to be valid.
  Expected output
  The query will remove the event from the *Simulation Event List.* It will roll back the simulation to the time prior to the occurrence of the event (The Rollback

algorithm will be described at the end of this section), start the simulation from that point on and display the history.

- *What if we insert an event?*:
  This query is posed when the user wants to analyse the consequences of inserting an event.
  Input from the user
  The user will enter an event, the TROM object on which this event occurs, the port on which this event will occur and the time this event will occur.
  Precondition:
  The entered event has to be an environmental event i.e. an output unconstrained event. TROM object and port have to be valid.
  Expected output
  The query will insert the new event in the *Simulation Event List* at the appropriate time. It will roll back the simulation to the time prior to the occurrence of the event. Start the simulation from that point on.

- *What if we reschedule an event?*:
  This query is posed when the user wants to analyse the consequences of rescheduling and event.
  Input from the user
  The user will enter an event, the TROM object on which this event occurs, the port on which this event will occur, the time this event occurred and the time this event is to occur.
  Precondition:
  The entered event cannot be an input event. The event cannot be an internal unconstrained event. If the event is output unconstrained there is no precondition on the timing. If the event is a constrained event the event's new time has to be within the timing interval allowed by the timing constraint. TROM object and port(in the case of an output event) have to be valid.
  Expected output
  The query will check if the new time is allowed. If allowed the event is rescheduled and the simulation is rolled back to the time prior to the min(old time, new time). Start the simulation from that point on. if not allowed the user is informed as to why it is not allowed.

Rollback Algorithm
If rollback time < 0 then error
else
for each TROM object
Set the current state to its state at Rollback time
Set the Assignment Vector *to its image at Rollback time*
Set the Reaction Vector *to its image at Rollback time*
Remove all the events whose occur time is > Rollback time
Reschedule all the output unconstrained events
whose occur time > Rollback time
Reschedule all the corresponding Rendezvous
Reschedule all the outstanding reaction
Schedule all unconstrained events from current state
set the simulation time to the Rollback time

6.3 Validation of the Specifications.

All the queries in this section are independent of the *Simulator*. They access the *AST* structure and analyse two different routing schemes. Every TROM object is represented by an augmented state machine diagram. All the TROM objects in the system are related to each other in the *SCS*. There are two types of routes:

1. Routes within a TROM object.

2. Routes within an *SCS*.

We define a route as the sequence of transitions that would lead a TROM object to go from one state to the other. All these routes have to be acyclic, i.e. they do not contain any cycles, otherwise we will have an infinite number of routes. We define the length of the route as the time interval representing the minimum and the maximum times needed to go from one state to the other. To calculate these timing intervals we will rely on the *Timing Constraints*. When we refer to time in this section we are referring to the *Simulation Time*, since in this section we do not consider the dynamic information of the *Simulator*; We are referring to absolute time. All these queries can be invoked at any time with or without running the simulation.

Use Case Analysis

This Use Case Diagram illustrated in Figure 26 contains three actors: the user, the *Reasoning System* and the *AST*. At any time after the *AST* is built, the user invokes a query in the *Reasoning System*. The *Reasoning System* then uses the methods that we created in the *AST* to find the routes. The *Reasoning System* then gives those results to the user.

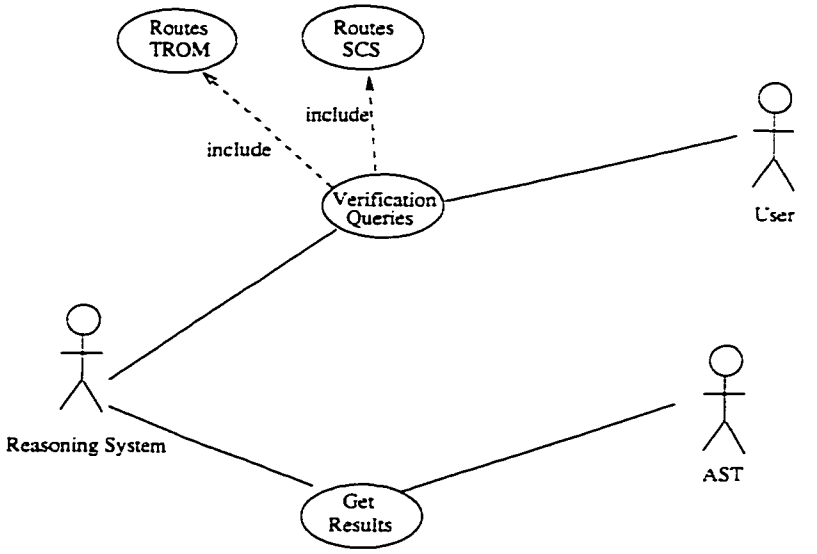


Figure 26: Use Case Diagram For Verification Tool

Validation of the Specifications Queries

In this section we will describe each query by showing the required input, the precondition and the expected output. We will describe the algorithm used to achieve the required results. The queries are the following:

- *Find all the routes between two states of a TROM object.* This query is posed when the user wants to see all the possible routes between any two states of any TROM object. The length of the route does not refer to other objects in the system, it is deduced from the *Timing Constraints* of the TROM object itself.
  Input from the user
  The user will enter the TROM objects, the initial state and the destination state.
  Precondition:

The entered TROM object has to be a valid TROM objects instantiated in the SCS. The corresponding states have to be valid states.

Expected output

The query will provide a list of all the acyclic routes that connect these two states in this TROM object. If there are no routes the query will tell the user the State is not reachable. By looking at all the routes the user can determine if there are any routes which are not desired. If there are any undesired routes or if a state is no reachable then the user will modify the design accordingly.

Algorithm

We used a depth first algorithm.[Shin92]

For Each transition going out of the initial state
Insert transition into dynamic route
Repeat until dynamic route = null
if destination state of transition = desired state
save route in outputted route list
remove last transition in route
else if transition causes a cycle
remove last transition from the dynamic route
else get the next transition going out of the destination
state of the last transition in the dynamic route
if no more transitions and route not empty
remove the last transition from the route
go back to the repeat
end of For
if the outputted route list is empty then display: no routes between S1 and S2.

- *Find one route to a specific state of a TROM object.*

  This is a reachability query. This query is posed when the user wants to see one route from the initial state to the destination state entered by the user. This route will take into consideration all the related TROM objects in the *SCS* and all the transitions within these TROM objects that are needed so that the TROM in question can reach the required State. The length of the route will depend on the *Timing Constraints* of all the TROM objects that are involved in this route.

Input from the user

The user will enter the TROM objects and the destination state.

Precondition:

We assume that there are no cycles in the SCS, otherwise the route we find may be infinite in length. The entered TROM object has to be a valid TROM objects instantiated in the SCS.The corresponding state has to be a valid state.

Expected output

The query will provide an acyclic route that connect the initial state to the state entered by the user. It will also show the routes needed in other related TROM objects to achieve this route. It is worth noting that the routes in related TROM objects may contain cycles. This may be needed. To have a better understanding of why this may be needed please refer to Figure 27. To go from state A to State D in TROM object A The input event I has to occur twice. This means that the route in TROM object be has to be I,B1,C1,I and this route contains a cycle. The TROM object may need two input events from a related TROM object and the corresponding output events may lead the related TROM object to have a cyclic route.

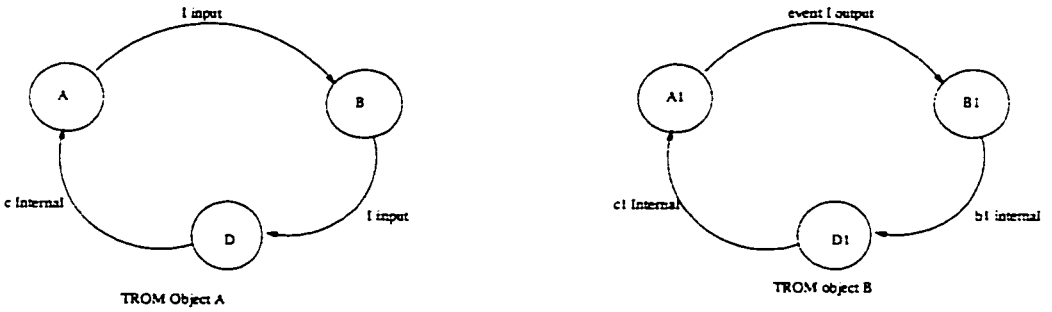


Figure 27: Need for cyclic routes in related TROM objects.

This query will also check if all the input events will occur. It may be the case that one of the events may not occur due to conflicting timing constraints. This is very helpful since this will detect timing inconsistencies in the design. An example of this type of query is illustrated in the *Robotics Assembly* example described in the next chapter. This query detected a timing inconsistency within the design of the *Robotics Assembly* example described in detail in the next section. The user by looking at the length of the route can deduce if this route

is feasible within a timing constraint. If there is a required input event that cannot occur du to the absence of the corresponding TROM object from the SCS then this query will tell the user that this state is not reachable. The user can modify the design accordingly.

Algorithm for finding a route to a state in a TROM object
find all the routes going from initial state to destination state save list1
take the first route from list1and insert it into a route list.
For each input event in the route
find the related TROM object.
if related TROM object is null then
move cursor on list1
if cursor = null
output state not reachable
return
find the cumulative route needed so that the corresponding output event would be fired.
insert this new route into the list
repeat what you did for the first node in the route list recursively for the other node in the route list until you find a node with no input events.
calculate the times for each route going from the last node in the route list to the first node.
check for timing inconsistencies.
Algorithm for findind cumulative route
if only one output event needed
find a route between the initial state and the state succeeding the necessary transition.
else
set state1 to initial state
for each event needed find the route between state1 and the state succeeding the necessary transition
set state1 to the state succeeding the last transition in the rout e
concatenate the route with the previous one.
return the concatenated route.

Chapter 7

Case Study : Robotics Assembly example

7.1 Introduction

This Chapter demonstrates the applications of the *Reasoning System* for a *Robotics Assembly* problem. The model of assembly will be described, first informally and then formally, and then the tool's application will be shown.

7.2 Problem Description

7.2.1 Informal Problem Description

We abstract *robots* from mechanical objects to functional units. The assembly environment consists of a *robot* with two arms, a *conveyer belt*, a *vision system*, and a *user*. A *user* places two kinds of *parts*, *cup* and *dish*, on the *belt*. The *vision system* senses a *part* on the *belt* and recognises its type. The *belt* stops whenever a *part* is sensed, so that the *robot* can pick the *part* from the *belt*. After the *part* is picked by the *robot*, the *belt* moves again. An assembly is performed when the *robot* matches a *cup* in one arm with a *dish* in the other arm. It is required to design the assembly system with real-time constraints, so that when n cups and n dishes are placed in an arbitrary ordering on the *belt*, n assemblies are made by the *robot*.

Constraints and Assembly Algorithm

The following assumptions are made:

- Both arms of the *robot* manipulator have the same physical characteristics (precision, speed, degrees of freedom) and functional capabilities.
- Algorithms for *part* recognition, collision-free motion of *robot* arms, gripping, holding, and placement work in real-time.
- The conveyor *belt* runs at a constant speed. No two parts can sit on the *belt* side by side nor can they collide while moving.

The following timing constraints must be specified:

1. There is a maximum delay of 2 time units from the instant a *part* enters the sensor zone on the *belt* to the instant it is sensed.
2. There is a maximum delay of 5 time units from the instant a *part* is sensed to the instant the vision system completes *part* recognition and informs the *robot*.
3. From the instant of receiving the signal from the vision system, the *robot* manipulator picks up the *part* from the *belt* within 2 time units.
4. To complete an assembly, the right arm should place the *part* it holds on the assembly pad, within a window of 2 to 4 time units of picking that *part*.

Our algorithm uses a *stack* to assemble the parts. Initially the left arm of the manipulator is free, the *stack* is empty, and no *part* has been sensed. Whenever both arms of the *robot* are free and the *stack* is empty, and a signal is received by the *robot* from the vision system, indicating the recognition of a *part*, the left arm picks up the *part* from the *belt*. If the left arm holds a *part* and the right arm is free at the instant the *part* recognition signal is received from the *vision system*, the right arm picks up the *part* from the *belt*. If both arms hold parts of the same kind the *part* in the right arm is pushed onto the *stack*; otherwise the parts are assembled as follows. The left arm places the *part* on the assembly tray and frees itself; next, the right arm places the *part* on the assembly tray. If the left arm is free and the right arm is not free, but the *stack* is not empty, the left arm picks up a *part* from the *stack*.

State	Left arm	Right arm
s1	free	free
s2	moving	free
s3	holding	free
s4	holding	moving
s5	holding	holding
s6	placing	holding
s7	holding	assembling
s8	holding	pushing on stack
s9	popping stack	holding

Table 1: States of Robot Manipulator.

Visual Models of a Design

We abstract the following components of the assembly unit: *User*, *belt*, *Vision System*, and *Robot*. The port types and messages among these components can be derived from the informal design description. Figure 28 shows the TROM classes, with respective port types in the *Robotics Assembly* system. We model each component as a GRC with port types and attributes. The *User* has one port type $@VS$ to communicate with the *VisionSystem* when parts are placed on the *belt*. The *belt* has two port types: port type $@V$ to receive a message from the *VisionSystem* when a *part* has been sensed; and port type $@R$ to receive messages from the *Robot* when a *part* has been picked. The *VisionSystem* has three port types: port type $@U$ to receive messages from the *User*; port type $@S$ to inform the *Robot* that a *part* has been recognised; and port type $@Q$ to inform the *belt* that a *part* has been sensed. The *Robot* has two port types: port type $@C$ to receive messages from the *VisionSystem* when a *part* has been recognised; and port type $@D$ to inform the *belt* that a *part* has been picked.

The dynamic behaviour of the reactive objects are captured in the state-chart diagrams shown in Figure 30, Figure 36, Figure 33, and Figure 39. The assembly system, consisting of two users, one *vision system*, one *belt*, and one *robot*, is described in the collaboration diagram in Figure 42. The formal specifications are shown in Figures 29, 35, 32, and 38. The LSL trait *PartType[Part]* is an abstract enumerated type for defining *cup* and *dish* parts. Table 7.2.1 describes the situations captured by the states for the robot manipulator in Figure 39.

The users place parts on the *belt* in an arbitrary order; however, the parts arrive in the sensor zone according to a first-in-first-out scheme. We capture this feature by introducing the attribute *inQueue* of type *PQueue*, where *Queue[Part,PQueue]* is an LSL trait defining a *queue* of parts. The attribute *inStack* of type *PStack*, where *Stack[Part,PStack]* is an LSL trait, models the operations of a *stack*. By including these traits in the GRCs, we have imported their operations into the formal specifications, thus abstracting the data computations. For instance, whenever the message *PutC* or *PutD* is received by the vision system, the corresponding *part* is enqueued. The parts are sensed and recognised in the order they are placed on the *belt*, subject to the timing constraints. This design ensures that every *part* placed on the *belt* is eventually recognised and assembled.

7.2.2 Class Diagram for Robotics Assembly

1. *Vision system* TROM class is an aggregate of port types @U, @S, @Q.
2. *User* TROM class is an aggregate of a port type @VS.
3. *Belt* TROM class is an aggregate of port types @R, @V.
4. *Robot* TROM class is an aggregate of port types @C, @D.

There is an association between the port type @Q of *Vision system* and @V of the *Belt*, meaning that the *Vision system* uses the port type @Q to communicate with the *Belt* through port type @V.
There is an association between the port type @U of *Vision system* and @VS of the *User*, meaning that the *Vision system* uses the port type @U to communicate with the *User* through port type @VS.
There is an association between the port type @S of *Vision system* and @C of the *Robot*, meaning that the *Vision system* uses the port type @S to communicate with the *Robot* through port type @C.
There is an association between the port type @D of *Robot* and @R of the *Belt*, meaning that the *Robot* uses the port type @D to communicate with the *Belt* through port type @R.
Vision system has two attributes, P of trait type *Part*, and *inQueue* of trait type *Queue*. The two types are abstract data types defined in the LSL traits *Part* and

Queue, where the *Queue* is parameterised by *Part*.

Robot has two attributes, P of trait type *Part*, and *inStack* of trait type *Stack*. The two types are abstract data types defined in the LSL traits *Part* and *Stack*, where the *Stack* is parameterised by *Part*.

The following figure shows the TROM classes, with respective port types in the Robotics assembly system.

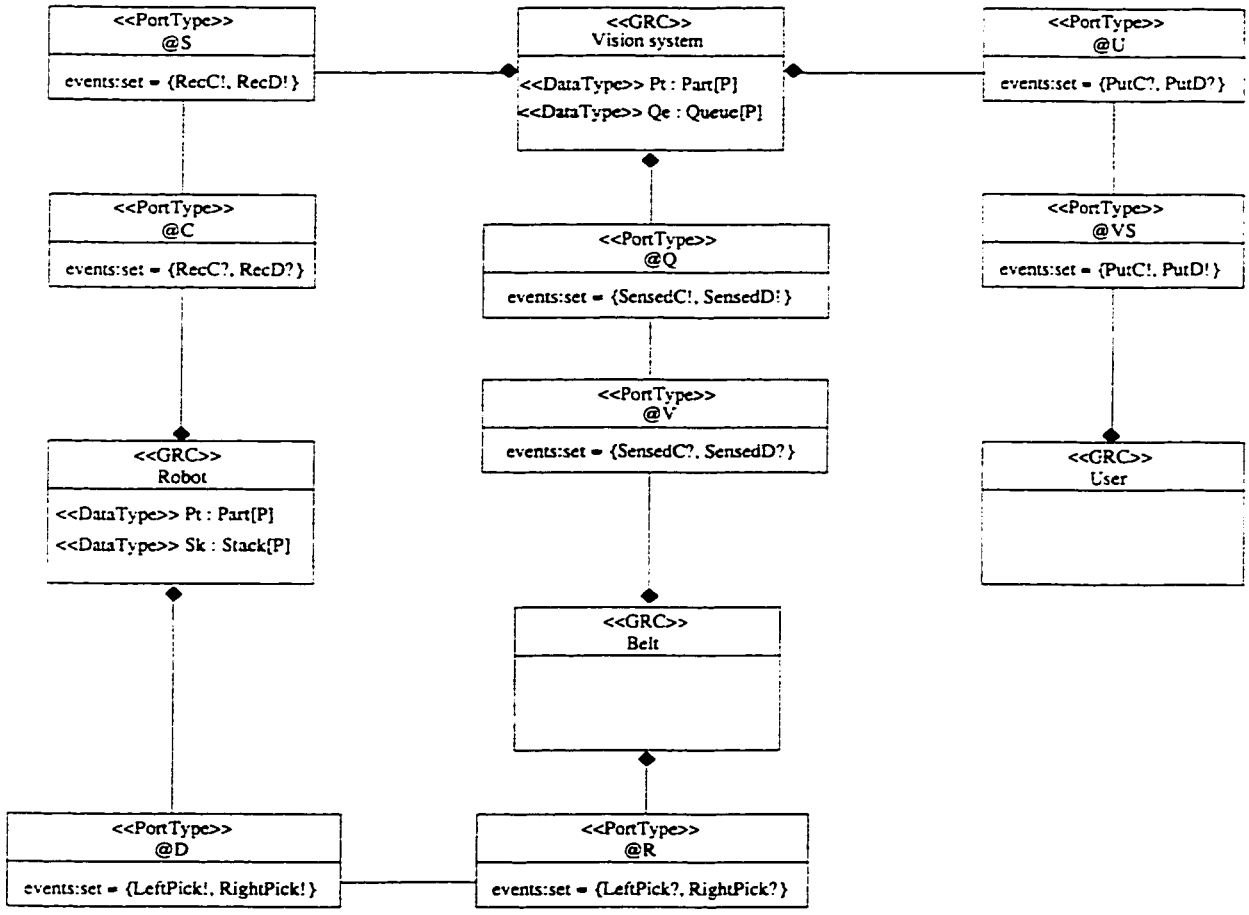


Figure 28: Robotics System Class diagram

7.2.3 Formal Problem Description

In this section we are going to describe each class in the Robotics assembly using three different notations namely, a textual representation which is used by the Interpreter to the build the internal structure i.e. the AST, the state machine representation, and the UML model developed using Rose tool. Following the description of the TROM

classes, we will be describing the LSL traits used in the Robotics assembly system. and the Subsystem configuration specification(SCS).

The User Class

The *User* is the only environmental class in the system, which controls the whole system by placing parts for assembly on the *belt*. Since the *User* is an environmental class, all its output events cannot be constrained by any other transitions.

Class User [@VS]

Events: Next,PutC!@VS, PutD!@VS, Resume
States: *idle, ready, place
Attributes:
Traits:
Attribute-Function: idle -> {}; ready -> {}; place -> {};
Transition-Specifications:
R1: <idle,ready> ; Next(true); true => true;
R2: <ready,place>; PutD(true); true => true;
R3: <ready,place>; PutC(true); true => true;
R4: <place,idle> ; Resume(true); true => true;
Time-Constraints:

end

Figure 29: User TROM class - Textual representation

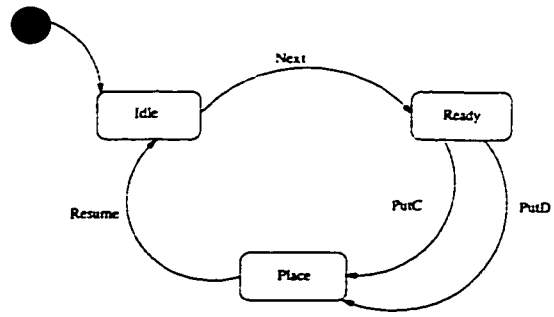


Figure 30: User TROM class - State machine representation

The Vision system Class

The *Vision system* communicates with the *User*, to know when a *part* is placed on the *Belt*. It inserts this *part* into the *Queue* and within certain time it will sense this

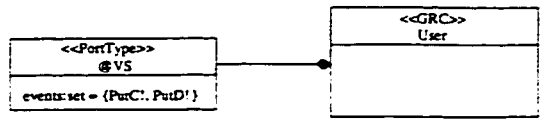


Figure 31: User TROM class - UML model

part and signals the *Belt* to stop moving. After a certain time it will signal the *Robot* to remove the *part* from the *Belt*. If during this time it receives another signal from the *User* and it has inserted the *part* into the *Queue* it will signal again the *Belt* to stop and *Robot* to pick that *part*, otherwise it will go into a *monitor* state.

Class Visionsystem [@U,@S, @Q]

Events: PutD?@U, PutC?@U,SensedD!@Q, SensedC!@Q,RecC!@S, RecD!@S
States: *monitor,active,identify
Attributes: inQueue:PQueue; P:PART
Traits: Part[PART], Queue[PART,PQueue]
Attribute-Function: monitor -> {inQueue}; active -> {inQueue}; identify -> {inQueue};
Transition-Specifications:
R1: <monitor,active> ; PutD(true) ; true => inQueue' = append(dish(P), inQueue);
R2: <monitor,active> ; PutC(true) ; true => inQueue' = append(cup(P), inQueue);
R3: <active,identify> ; SensedD(true); head(inQueue)=dish(P) => true;
R4: <active,identify> ; SensedC(true); head(inQueue)=cup(P) => true;
R5: <active,active> ; PutD(true) ; true => inQueue' = append(dish(P),inQueue);
R6: <active,active> ; PutC(true) ; true => inQueue' = append(cup(P),inQueue);
R7: <identify,monitor> ; RecC(true) ; len(inQueue) = 1 => inQueue' = tail(inQueue);
R8: <identify,identify>; PutD(true) ; true => inQueue' = append(dish(P),inQueue);
R9: <identify,monitor> ; RecD(true) ; len(inQueue) = 1 => inQueue' = tail(inQueue);
R10:<identify,active> ; RecD(true) ; len(inQueue) > 1 => inQueue' = tail(inQueue);
R11:<identify,active> ; RecC(true) ; len(inQueue) > 1 => inQueue' = tail(inQueue);
R12:<identify,identify>; PutC(true) ; true => inQueue' = append(cup(P),inQueue);
Time-Constraints:
TC1: R2, SensedC, [0,2],{};
TC2: R1, SensedD, [0,2],{};
TC3: R4, RecC, [0,5],{};
TC4: R3, RecD, [0,5],{};
TC5: R10,SensedC, [0,2],{};
TC6: R10,SensedD, [0,2],{};
TC7: R11,SensedC, [0,2],{};
TC8: R11,SensedD, [0,2],{};
end

Figure 32: Vision system TROM class - Textual representation

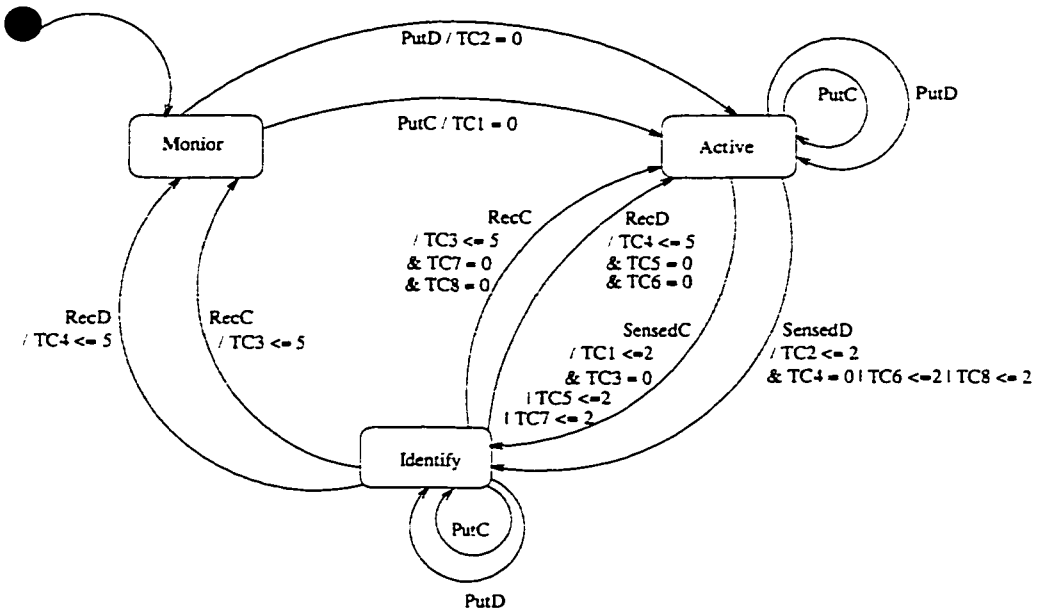


Figure 33: Vision system TROM class - State machine representation

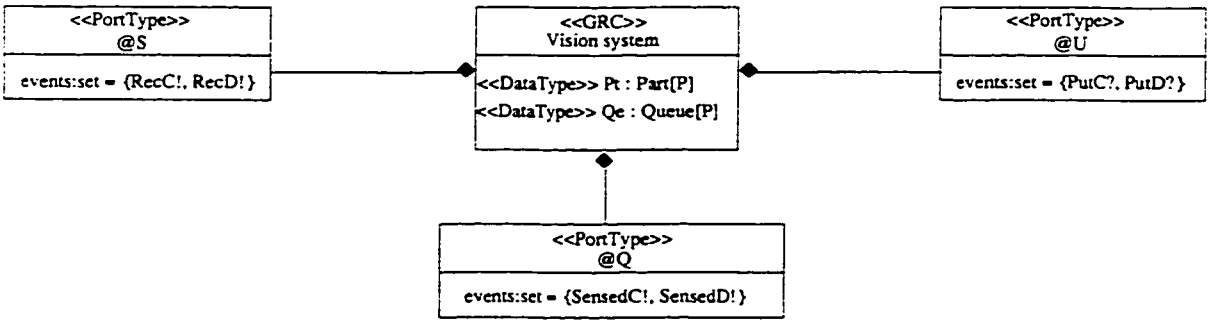


Figure 34: Vision system TROM class - UML model

The Belt Class

The *Belt* is controlled by both the *Vision system* and the *Robot*. It will stop whenever the *Vision system* senses a *part*, and starts moving again whenever the *Robot* picks the *part* up.

Class Belt [@R,@V]

Events: SensedC?@V, SensedD?@V, LeftPick?@R, RightPick?@R
States: *active,stop
Attributes:
Traits:
Attribute-Function: active -> {}; stop -> {};
Transition-Specifications:
R1: <active,stop> ; SensedC(true) ; true => true;
R2: <active,stop> ; SensedD(true) ; true => true;
R3: <stop,active> ; LeftPick(true) ; true => true;
R4: <stop,active> ; RightPick(true); true => true;
Time-Constraints:

end

Figure 35: Belt TROM class - Textual representation

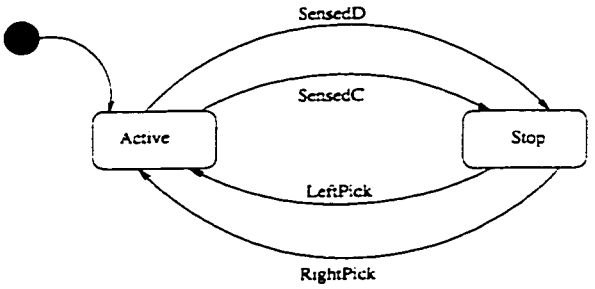


Figure 36: Belt TROM class - State machine representation

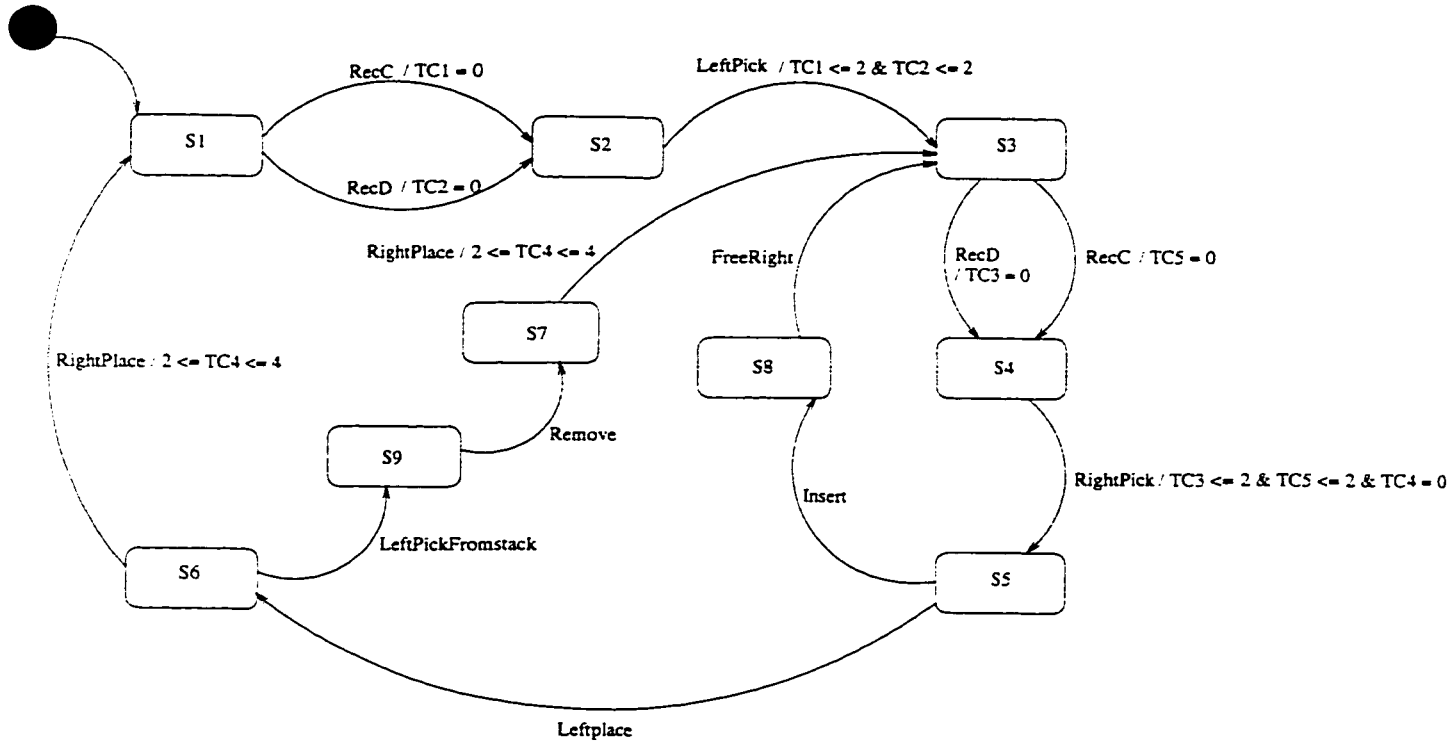
Figure 37: Belt TROM class - UML model

The Robot Class

The *Robot* has two manipulators namely the left and the right arm. Whenever an arm picks up a *part* it signals the *Belt* to start moving again. The left arm will pick up the first *part* followed by the right arm. If there are of the same type, the right arm will insert the *part* it has into a *Stack* and wait to pick up another *part*. If they are not the same, the left arm will start the assembly by placing the *part* it has on the tray. It will then check to see whether there are any *parts* in the *stack*, if there is a *part* then it picks it from the *stack* and the right arm will then finish the assembly by placing the *part* on the tray. If there are no parts in the *stack*, the right arm will finish the assembly, and both arms will be free.

```
Class Robot [@D,@C]
    Events:          RecC?@C, RecD?@C, LeftPick!@D, RightPlace, Remove,
                     RightPick!@D, LeftPlace, Insert, FreeRight, LeftPickFromStack
    States:          *S1, S2, S3, S4, S5, S6, S7, S8, S9
    Attributes:      lPrt:PART; rPrt:PART; inStack:PStack
    Traits:          Part[PART], Stack[PART, PStack]
    Attribute-Function: S1 -> {}; S2 -> {lPrt}; S6 -> {}; S7 -> {lPrt,inStack}; S3 -> {inStack}; S4 ->{rPrt}; S5 -> {};
    Transition-Specifications:
      R1: <S1,S2> ; RecC(true) ; true  => lPrt' = cup(lPrt);
      R2: <S1,S2> ; RecD(true) ; true  => lPrt' = dish(lPrt);
      R3: <S2,S3> ; LeftPick(true) ; true => true;
      R4: <S6,S1> ; RightPlace(true) ; isEmpty(inStack) => rPrt' = nullpart(rPrt);
      R5: <S6,S9> ; LeftPickFromStack(true); !(isEmpty(inStack)) => lPrt' = top(inStack);
      R6: <S7,S3> ; RightPlace(true); true  => rPrt = nullpart(rPrt);
      R7: <S3,S4> ; RecC(true) ; true  => rPrt' = cup(rPrt);
      R8: <S3,S4> ; RecD(true); true => rPrt' = dish(rPrt);
      R9: <S4,S5>; RightPick(true); true => true;
      R10:<S5,S6>; LeftPlace(true) ; !(lPrt = rPrt) => lPrt' = nullpart(lPrt);
      R11:<S5,S8> ; Insert(true); rPrt = lPrt => inStack' = push(rPrt, inStack);
      R12:<S8,S3>; FreeRight(true); true => rPrt' = nullpart(rPrt);
      R13:<S9,S7>; Remove(true) ; true => inStack' = pop(inStack);
    Time-Constraints:
      TC1: R1, LeftPick,  [0,2], {};
      TC2: R2, LeftPick,  [0,2], {};
      TC3: R8, RightPick, [0,2], {};
      TC4: R9, RightPlace, [2,4], {};
      TC5: R7, RightPick, [0,2], {};
end
```

Figure 38: Robot TROM class - Textual representation



S1 - Both Arm are Free
S2 - Left Arm ready to pick, Right Arm free
S3 - Left Arm not free, Right Arm free
S4 - Right Arm ready to pick, Left Arm not free
S5 - Right Arm not free, Left Arm not free
S6 - Left Arm is free, Right Arm is not free
S7 - Right Arm ready to place, Left Arm not free
S8 - Right Arm inserting into Stack, Left Arm not free
S9 - Left Arm removing from stack, Right Arm is not free

Figure 39: Robot TROM class - State machine representation

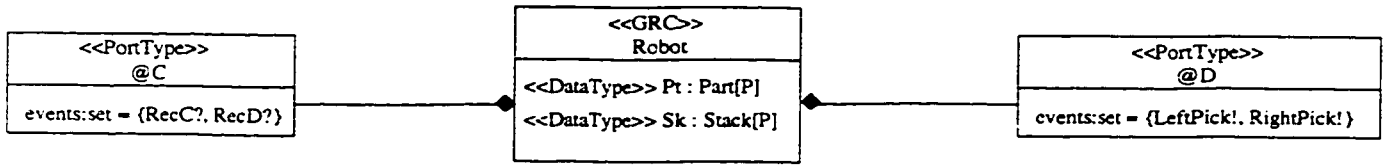


Figure 40: Robot TROM class - UML model

The Subsystem Configuration Specification(SCS)

The system we are going to simulate is composed of one *Robot*, one *Belt*, one *User*, and one *Vision system*. Figure 41 shows the textual representation, and Figure 42 shows the UML model of the SCS.

```
SCS Robot
    Includes:
    Instantiate:
            r1::Robot[@D:1, @C:1];
            b1::Belt[@R:1, @V:1];
            u1::User[@VS:1];
            v1::Visionsystem[@U:1, @S:1, @Q:1];
    Configure:
            u1.@VS1:@VS <-> v1.@U1:@U;
            b1.@V1:@V <-> v1.@Q1:@Q;
            v1.@S1:@S <-> r1.@C1:@C;
            r1.@D1:@D <-> b1.@R1:@R;

end
```

Figure 41: SCS - Textual representation

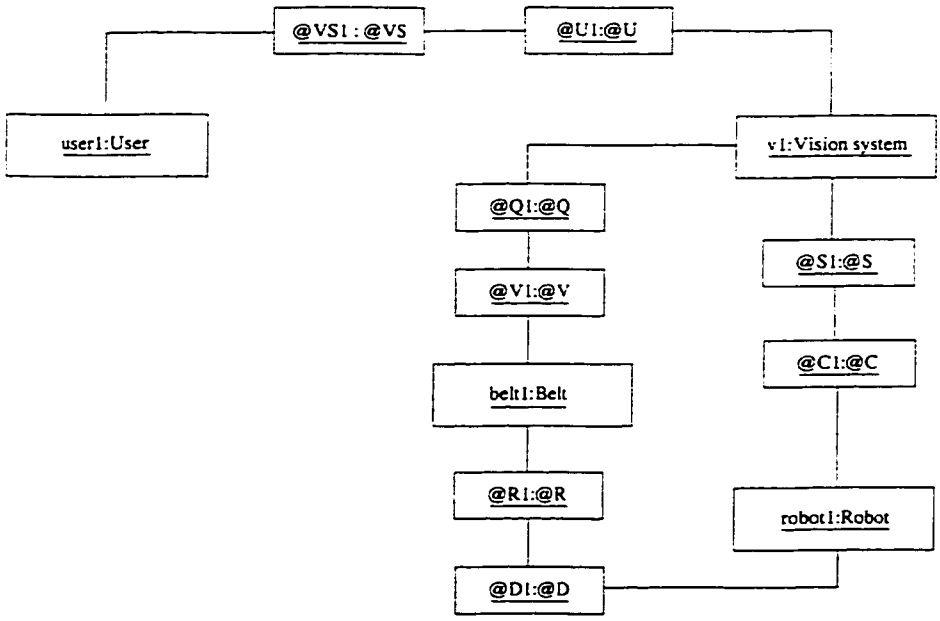


Figure 42: SCS - UML model

Sample Simulation Event List

In the sample simulation event list we will schedule four events namely *PutC*,*PutC*,*PutD*, and *PutD* of the *User* object which will be instantiated in the SCS. This is due the fact that only output unconstrained events i.e. environmental events are allowed in the initial simulation event list. All subsequent events will be scheduled by the *Simulator* as the simulation proceeds. Figure 43 shows the textual representation of the sample *Simulation Event List.*

```
SEL: Robot
   PutD, u1, @VS1, 3;
   PutD, u1, @VS1, 5;
   PutC, u1, @VS1, 7;
   PutC, u1, @VS1, 9;
end
```

Figure 43: Sample Simulation Event List

LSL Traits

The system uses the three LSL traits: *Part*, *Queue*, and *Stack*. The *Part* trait is used by the *Vision system* and the *Robot*. The *Queue* trait as mentioned earlier will be used by the *Vision system* to store the parts placed on the *Belt*. The *Stack* trait is used by the *Robot* to push and pop the parts as mentioned in the previous section. Figures 44, 45 and 46 show the textual representation of three traits namely *Part*, *Queue* and *Stack* respectively.

```
Trait: Part(P)
   Includes: Boolean
   Introduce:
    cup   : P -> P;
    dish  : P -> P;
    free  : P -> P;
end
```

Figure 44: Part LSL Trait

```
Trait: Queue(e, Q)
   Includes: Integer
   Introduce:
     insert : e, Q -> Q;
     delete : Q    -> Q;
     head   : Q    -> e;
     size   : Q    -> Int;
end
```

Figure 45: Queue LSL Trait

```
Trait: Stack(e, S)
   Includes: Boolean
   Introduce:
     isEmpty: S -> bool:
     push :  e, S -> S;
     pop :       S -> S;
     top :       S -> e;
end
```

Figure 46: Stack LSL Trait

7.3 Reasoning on the Robotics Assembly

In this section we give the results of the diffrent types of queries that were discussed in the previous chapter. This section is divided into three subsections, one for each type of reasoning:

- The *Reasoning System* as a Debugging tool;
- The *Reasoning System* for Hypothetical Queries;
- Using the *Reasoning System* for Validation of the Specifications:

We will show in these subsections the syntax to pose the queries and the results yielded by the *Reasoning System*.

7.3.1 The *Reasoning System* as a Debugging tool

In this section we will show all the queries that will be used to help the user in the debugging process. These queries will help the user to have a better understanding of the results of the simulation. They will offer the user different perspectives on the

history. The input to these queries and the results they give are described in detail in this section.

Why did the system go from state S1 to state S2 - an Example

When the simulation is stopped in debugging mode the user invokes the *Reasoning System*. The user then selects the Query.

This is a sample dialogue with the user:

Please enter trom label: u1
Please enter the initial state: idle
Please enter the final state: idle
Are there any more trom objects: n

This means the user wants to know why the TROM object *u1*(the user in the *Robotics Assembly Example*) went from the *idle* state to the *idle* state during the simulation.

If this query is asked at time 4 we get the following output:

Simulation Events between state idle and state idle:
Sim-Event 1:
Simulation Event : Next Trom : u1
Time : 0
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : idle
Assignment Vector prior to transition :
AssignmentVector :
Event Consequence :
cause : internal unconstrained event
Sim-Event 2:
Simulation Event : PutD Trom : u1
Port : VS1 Time : 3
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : ready

Assignment Vector prior to transition :
AssignmentVector :
Event Consequence :
cause : event entered by the user
Sim-Event 3:
Simulation Event : Resume Trom : u1
Time : 3
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : place
Assignment Vector prior to transition :
AssignmentVector :
Event Consequence :
cause : internal unconstrained event
Sim-Event 4:
Simulation Event : Next Trom : u1
Time : 3
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : idle
Assignment Vector prior to transition :
AssignmentVector :
Event Consequence :
cause : internal unconstrained event

Result:

This tells the user that the TROM object *u1* went from state *idle* to state *idle* at time 3 because of the following dynamics:

- The simulation event *next* occurred at time 0, the cause of this event is: this event is an internal unconstrained event.
- The simulation event *PutD* occurred at time 3, the cause of this event is: this event is an output unconstrained event that was entered by the user.
- The simulation event *next* occurred at time 3, the cause of this event is: this

event is an internal unconstrained event

This query gives the user a better understanding of the behaviour of particular TROM objects. It isolates that particular object from the complex system and gives all the events and their reasons.

Display the simulation event list of a TROM Object - an Example

When the simulation is stopped in debugging mode the user invokes the *Reasoning System.* The user then selects the Query.

This is a sample dialogue:

Please enter trom label: v1

This means the user wants to know all the simulation events that are related to the TROM object *v1* until the current time of the simulation.

If this query is asked at time 4 we get the following output we get:

Simulation Events between 0 and 4:
Sim-Event 1:
Simulation Event : PutD Trom : v1
Port : U1 Time : 3
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : monitor
Assignment Vector prior to transition :
AssignmentVector :
Attribute Name : inQueue
Trait Type
Trait : Trait type : PQueue Trait name : Queue
Trait value:
Queue : Size -> 0
Attribute Name : P
Trait Type
Trait : Trait type : PART Trait name : Part
Trait value:

Event Consequence :
Time Constraint:
Time Constraint: TC2 Event: SensedD
Reaction Window:
3,5
Outcome: ENABLED
cause : PutD Trom u1 time 3
Sim-Event 2:
Simulation Event : SensedD Trom : v1
Port : Q1 Time : 4
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : active
Assignment Vector prior to transition :
AssignmentVector :
Attribute Name : inQueue
Trait Type
Trait : Trait type : PQueue Trait name : Queue
Trait value:
Queue : Size -> 1
dish
Attribute Name : P
Trait Type
Trait : Trait type : PART Trait name : Part
Trait value:
dish
Event Consequence :
Time Constraint:
Time Constraint: TC2 Event: SensedD
Reaction Window:
3,5
Outcome: FIRED
Time Constraint:
Time Constraint: TC4 Event: RecD

Reaction Window:
4.9
Outcome: ENABLED
cause : PutD Trom v1 time 3
Sim-Event 3:
Simulation Event : RecD Trom : v1
Port : S1 Time : 4
History:
Event Outcome : NOTYET HANDLED
cause : SensedD Trom v1 time 4
End of Simulation Event List.

Result:

This is a list of all the events the assignment vector and the reaction vector that are related to TROM *v1* from time 0 to time 4, time 4 being the current simulation time. This query gives the user a better understanding of the behaviour of particular TROM objects. It isolates that particular object from the complex system and gives all the events and the reasons for their occurrence.

When Query - an Example

When the simulation is stopped in debugging mode the user invokes the *Reasoning System.* The user then selects the Query *When.* This query in turn leads to a selection among six possible queries.

1. *When was the system in state 1?*

   This is a sample dialogue for the first sub-query:

   Please enter trom label: u1
   Please enter the state: idle
   are there any more trom objects: y
   Please enter trom label: v1
   Please enter the state: active
   are there any more trom objects: n

   This means that the user wants to know during which time intervals the Trom object *u1* was in state *idle* and at the same time the TROM object *v1* was in

state active.

If this query is asked at simulation time 9 then depending on the simulation, one of the outputs is shown below:

u1
time interval : 0 to 0
time interval : 3 to 3
time interval : 5 to 5
time interval : 5 to 7
time interval : 7 to 9
v1
time interval : 3 to 4
time interval : 5 to 6
time interval : 9 to 9
The system is in that state at the following time interval(s)
time interval : 3 to 3
time interval : 5 to 5
time interval : 9 to 9

Result:

The TROM object *u1* was in the required state from time 0 to time 0, from time 3 to time 3 ,from time 5 to time 5,from time 7 to time 7 and from time 9 to time 9. The TROM object *v1* was in that state from time 3 to time 4, from time 5 to time 6 and from time 9 to time 9. The intersection of these timing intervals being 3 to 3 and 5 to 5 and 9 to 9.

This analyses can be very tough if the system is composed of many TROM objects. It is very helpful, the user can determine if any two or more objects were in states during a time interval and they were not supposed to be in those states.

2. *When did the system go out of state 1?*

This is a sample dialogue for the second sub-query:

Please enter trom label: u1
Please enter the initial state: idle

are there any more trom objects: y
Please enter trom label: v1
Please enter the initial state: active
are there any more trom objects: n

We are asking this query with the same data as we did for the previous query.
If this query is asked at simulation time 9 we get the following output:

u1
time interval : 0 to 0
time interval : 3 to 3
time interval : 5 to 5
time interval : 7 to 7
time interval : 9 to 9
v1
time interval : 3 to 4
time interval : 5 to 6
The system goes out of this state at the following time(s)
time : 3
time : 5

Result:

Since the TROM object *v1* at time 9 is still in the state *active* the last time interval that we had in the previous query i.e. 9 to 9 is not here and the result is that the system went out of the state at times 3, and 5
This query allows the user to see if the TROM objects went out of the states before or after they were supposed to.

3. *When the event was fired?*

This is a sample dialogue for the third sub-query:

Please enter event name: PutC
Do you want to enter a trom object (default is the entire system)(y/n): n
Attempting to see when event PutC was fired for all the system

That is the user wants to know when the event *PutC* was fired for the entire

system until the current time.

If this query is asked at time 9 the result is the following.

For trom u1 the event PutC was fired at time 7
For trom v1 the event PutC was fired at time 7
For trom u1 the event PutC was fired at time 9
For trom v1 the event PutC was fired at time 9

Result:

Event *PutC* was fired 4 times. Two output unconstrained events at TROM object *u1* and two corresponding rendezvous at TROM object *v1*.
This query will give the user a view of all the occurrences of a specific event. It gives a different point of view from the simulation results.

4. *When the event was disabled?*

This is a sample dialogue for the fourth sub-query.

Please enter event name: SensedC
Do you want to enter a trom object (default is the entire system)(y/n): y
Please enter trom label: v1
are there any more trom objects: n

This means the user wants to see if the event *SencedC* in the TTROM object *v1* was disabled at any time during the simulation until the current time.

We get the following output if we ask this query at time 10:

For trom v1 The event SensedC was disabled at time 6

Result:

If you analyse the *Robotics Assembly Example* carefully, you see that after receiving the first event *PutD* the event *SencedC* will be scheduled due to the timing constraint *TC5*, but this event will not be fired because the precondition is false; that is the first element in the queue is a *dish* and hence the event is disabled at time 6 when we attempt to fire it.

This query will tell the user if an event that was not supposed to be disabled was disabled. He can then use the *Why Query* to understand what happened, and may modify the design so that this will not happen again.

5. *When the event was enabled?*

   This is a sample dialogue for the fifth sub-query (when the event that caused this event to be scheduled was fired)

   Please enter event name: SensedD
   Do you want to enter a trom object (default is the entire system)(y/n):
   Attempting to see when event SensedD was enabled for all the system
   The user wants to know when the event *SensedD* was enabled for the entire system.

   The following output is obtained if we ask this query at time 10.

   For trom v1 The event SensedD was enabled at time 3
   For trom b1 The event SensedD was enabled at time 3
   For trom v1 The event SensedD was enabled at time 5
   For trom b1 The event SensedD was enabled at time 6
   For trom v1 The event SensedD was enabled at time 9
   For trom b1 The event SensedD was enabled at time 10

   Result:

   The time of the enabling of the event *SensedD* for the TROM object *b1* and the TROM object *v1* are not the same , *SensedD* was enabled for the em Vision System at time five but it only fired at time 6 thus enabling the event em SensedD on the *belt* at that time.

   This query will tell the user when an event was enabled. If we combine this query with the *Why Query* we can know when it was enabled and why it was enabled. This gives the user a better understanding of what is happening in the simulation.

6. *When the event was scheduled?*

   This is a sample dialogue for the sixth sub-query(at what time the event was scheduled to be fired)

Please enter event name: SensedD

Do you want to enter a trom object (default is the entire system)(y/n):

Attempting to see when event SensedD was scheduled for all the system

The user wants to know when the event *SensedD* was scheduled to be fired for the entire system.

The following output is obtained when we ask this query at time 10.

For trom v1 The event SensedD was scheduled at time 3

For trom b1 The event SensedD was scheduled at time 3

For trom v1 The event SensedD was scheduled at time 6

For trom b1 The event SensedD was scheduled at time 6

For trom v1 The event SensedD was scheduled at time 10

Result:

This response confirms the results we had in the previous query.

Show the assignment vector at a specific time

When the simulation is stopped in debugging mode the user invokes the *Reasoning System.* The user then selects the Query.

This is a sample dialogue for this query:

Enter the time for which you want to see the assignment vector 5

That is we want to see the assignment vector for the entire system at time 5.

The following output is obtained when we invoke this query at simulation time 10.

for trom : r1

AssignmentVector :

Attribute Name : lPrt

Trait Type

Trait : Trait type : PART Trait name : Part

Trait value:

dish

Attribute Name : rPrt

Trait Type

Trait : Trait type : PART Trait name : Part

Trait value:

Attribute Name : inStack
Trait Type
Trait : Trait type : PStack Trait name : Stack
Trait value:
Stack : Size -> 0
for trom : b1
AssignmentVector :
for trom : u1
AssignmentVector :
for trom : v1
AssignmentVector :
Attribute Name : inQueue
Trait Type
Trait : Trait type : PQueue Trait name : Queue
Trait value:
Queue : Size -> 1
dish
Attribute Name : P
Trait Type
Trait : Trait type : PART Trait name : Part
Trait value:
dish

Result:

The result we have is the status of the *assignment vector* for each TROM object of the system at time 5.

Without the *Reasoning System* the reconstruction of the *Assignment Vector* is not possible. By having an image of the assignment vector at a particular time we can determine the reason behind the disabling of an event, or the firing of an event. If this query is combined with one of the *When queries*, the user can have a better understanding of the results of the simulation.

Show the reaction vector at a specific time.

When the simulation is stopped in debugging mode the user invokes the *Reasoning System.* The user then selects the Query.

This is a sample dialogue for this query:

Enter the time for which you want to see the reaction vector 7

The user wants to see the reaction vector at time 7 for the entire system

The following output is obtained when we invoke this query at simulation time 10.

for trom : r1
Reaction Vector :
Reaction Sub Vector :
Time - Constraint : TC1
Reaction Sub Vector :
Time - Constraint : TC2
Reaction Sub Vector :
Time - Constraint : TC3
Reaction Sub Vector :
Time - Constraint : TC4
Reaction Sub Vector :
Time - Constraint : TC5
for trom : b1
Reaction Vector :
for trom : u1
Reaction Vector :
for trom : v1
Reaction Vector :
Reaction Sub Vector :
Time - Constraint : TC1
Reaction Sub Vector :
Time - Constraint : TC2
Reaction Sub Vector :
Time - Constraint : TC3

Reaction SubVector :

Time - Constraint : TC4

6,11

Reaction SubVector :

Time - Constraint : TC5

Reaction SubVector :

Time - Constraint : TC6

Reaction SubVector :

Time - Constraint : TC7

Reaction SubVector :

Time - Constraint : TC8

Result:

The result is an image of the *Reaction Vector* for each TROM object of the system at time 7.

Since the simulation tool does not keep the entire *Reaction Vector*, to reconstruct an image of this vector at a specific time would be very hard.

Does the system go into a specific state

When the simulation is stopped in debugging mode the user invokes the *Reasoning System.* The user then selects the Query.

This is a sample dialogue for this query:

Please enter trom label: u1

Please enter the initial state: ready

are there any more trom objects: n

The user wants to know if the TROM object *u1* went into the state *ready* at any time during the simulation.

If this query is asked at simulation time 3 we get the following output:

u1

time interval : 0 to 3

Yes The system is in that state at the following time interval(s)

time interval : 0 to 3

Result:

The TROM object *u1* is in state *ready* from time 0 to time 3

This query is similar to the query *"When the system was in state 1"*. It is essentially a different way of asking the same question.

Does the system go into a specific state more then once

When the simulation is stopped in debugging mode the user invokes the *Reasoning System.* The user then selects the Query.
This is a sample dialogue:

Please enter trom label: u1
Please enter the initial state: ready
are there any more trom objects: n

The user wants to know if the TROM object *u1* went into the state ready at any time during the simulation more than once.
We get the following output if we ask this query at simulation time 3:
u1
time interval : 0 to 3
No The system was not in that state more then once
Result:
TROM object *u1* is not in that state more than once during the simulation.

Show TROM status in a given time interval.

When the simulation is stopped in debugging mode the user invokes the *Reasoning System.* The user then selects the Query.
This is a sample dialogue:
Please enter trom label: v1
Please enter the lower time bound: 4
Please enter the upper time bound: 6

The user wants to know the status of the TROM object *v1* at every time between time 4 and time 6.
We get the following output if we ask this query at simulation time 10:

for time: 4
State name : identify
Initial state : false
AssignmentVector :
Attribute Name : inQueue
Trait Type
Trait : Trait type : PQueue Trait name : Queue
Trait value:
Queue : Size -> 1
dish
Attribute Name : P
Trait Type
Trait : Trait type : PART Trait name : Part
Trait value:
dish
for time: 5
State name : identify
Initial state : false
AssignmentVector :
Attribute Name : inQueue
Trait Type
Trait : Trait type : PQueue Trait name : Queue
Trait value:
Queue : Size -> 2
dish
dish
Attribute Name : P
Trait Type
Trait : Trait type : PART Trait name : Part
Trait value:
dish
for time: 6
State name : identify
Initial state : false

AssignmentVector :
Attribute Name : inQueue
Trait Type
Trait : Trait type : PQueue Trait name : Queue
Trait value:
Queue : Size -> 2
dish
dish
Attribute Name : P
Trait Type
Trait : Trait type : PART Trait name : Part
Trait value:
dish

Result:

The status of the TROM object at each discreet time in the given time interval.

7.3.2 Hypothetical Queries

In this section we will describe the hypothetical queries that will allow the user to have more control on the simulation scenario. These queries will give the user more control on the events and their timing. We will give first the type of input required, then we will show the Simulation Event List before and after the execution of the query, and we will show the simulation time, before and after the query.
When the simulation is stopped in debugging mode the user can invoke the *What if?* query. This set of hypothetical queries helps the user in testing new scenarios. The user can either insert a new event, remove an existing event or reschedule an existing event. This set of queries will modify the simulation results by changing the Simulation Event List. Once the changes are done the query will start running the *Simulator*.

What if we remove an event ?

Since the user represents the environment the user is only allowed to remove environmental events. All the other events are reactions to stimulus and the user is not allowed to remove them.

This is a sample dialogue with the user:

Enter event label: PutD

Enter trom label: u1

Enter port label: VS1

Enter occur time: 3

If this query is invoked at time 5 the *Simulation Event List* **prior to this query is the following:**

Simulation Event List:

Sim-Event 1:

Simulation Event : Next Trom : u1

Time : 0

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : idle

Assignment Vector prior to transition :

AssignmentVector :

Event Consequence :

cause : internal unconstrained event

Sim-Event 2:

Simulation Event : PutD Trom : u1

Port : VS1 Time : 3

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : ready

Assignment Vector prior to transition :

AssignmentVector :

Event Consequence :

cause : event entered by the user

Sim-Event 3:

Simulation Event : Resume Trom : u1

Time : 3

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : place

Assignment Vector prior to transition :
AssignmentVector :
Event Consequence :
cause : internal unconstrained event
Sim-Event 4:
Simulation Event : Next Trom : u1
Time : 3
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : idle
Assignment Vector prior to transition :
AssignmentVector :
Event Consequence :
cause : internal unconstrained event
Sim-Event 5:
Simulation Event : PutD Trom : v1
Port : U1 Time : 3
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : monitor
Assignment Vector prior to transition :
AssignmentVector :
Attribute Name : inQueue
Trait Type
Trait : Trait type : PQueue Trait name : Queue
Trait value:
Queue : Size -> 0
Attribute Name : P
Trait Type
Trait : Trait type : PART Trait name : Part
Trait value:
Event Consequence :
Time Constraint:
Time Constraint: TC2 Event: SensedD

Reaction Window:

3,5

Outcome: ENABLED

cause : PutD Trom u1 time 3

Sim-Event 6:

Simulation Event : SensedD Trom : v1

Port : Q1 Time : 4

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : active

Assignment Vector prior to transition :

AssignmentVector :

Attribute Name : inQueue

Trait Type

Trait : Trait type : PQueue Trait name : Queue

Trait value:

Queue : Size -> 1

dish

Attribute Name : P

Trait Type

Trait : Trait type : PART Trait name : Part

Trait value:

dish

Event Consequence :

Time Constraint:

Time Constraint: TC2 Event: SensedD

Reaction Window:

3,5

Outcome: FIRED

Time Constraint:

Time Constraint: TC4 Event: RecD

Reaction Window:

4,9

Outcome: ENABLED

cause : PutD Trom v1 time 3

Sim-Event 7:

Simulation Event : SensedD Trom : b1

Port : V1 Time : 4

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : active

Assignment Vector prior to transition :

AssignmentVector :

Event Consequence :

cause : SensedD Trom v1 time 4

Sim-Event 8:

Simulation Event : PutD Trom : u1

Port : VS1 Time : 5

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : ready

Assignment Vector prior to transition :

AssignmentVector :

Event Consequence :

cause : event entered by the user

Sim-Event 9:

Simulation Event : Resume Trom : u1

Time : 5

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : place

Assignment Vector prior to transition :

AssignmentVector :

Event Consequence :

cause : internal unconstrained event

Sim-Event 10:

Simulation Event : Next Trom : u1

Time : 5

History:
Event Outcome : NOTYET HANDLED
cause : internal unconstrained event
Sim-Event 11:
Simulation Event : PutD Trom : v1
Port : U1 Time : 5
History:
Event Outcome : NOTYET HANDLED
cause : PutD Trom u1 time 5
Sim-Event 12:
Simulation Event : PutC Trom : u1
Port : VS1 Time : 7
History:
Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 13:
Simulation Event : PutC Trom : v1
Port : U1 Time : 7
History:
Event Outcome : NOTYET HANDLED
cause : PutC Trom u1 time 7
Sim-Event 14:
Simulation Event : RecD Trom : v1
Port : S1 Time : 8
History:
Event Outcome : NOTYET HANDLED
cause : SensedD Trom v1 time 4
Sim-Event 15:
Simulation Event : RecD Trom : r1
Port : C1 Time : 8
History:
Event Outcome : NOTYET HANDLED
cause : RecD Trom v1 time 8
Sim-Event 16:

Simulation Event : PutC Trom : u1
Port : VS1 Time : 9
History:
Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 17:
Simulation Event : PutC Trom : v1
Port : U1 Time : 9
History:
Event Outcome : NOTYET HANDLED
cause : PutC Trom u1 time 9
End of Simulation Event List.

The Simulation Event List after the query is invoked is the following.

Simulation Event List:
Sim-Event 1:
Simulation Event : Next Trom : u1
Time : 0
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : idle
Assignment Vector prior to transition :
AssignmentVector :
Event Consequence :
cause : internal unconstrained event
Sim-Event 2:
Simulation Event : PutD Trom : u1
Port : VS1 Time : 5
History:
Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 3:
Simulation Event : PutD Trom : v1
Port : U1 Time : 5

History:
Event Outcome : NOTYET HANDLED
cause : PutD Trom u1 time 5
Sim-Event 4:
Simulation Event : PutC Trom : u1
Port : VS1 Time : 7
History:
Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 5:
Simulation Event : PutC Trom : v1
Port : U1 Time : 7
History:
Event Outcome : NOTYET HANDLED
cause : PutC Trom u1 time 7
Sim-Event 6:
Simulation Event : PutC Trom : u1
Port : VS1 Time : 9
History:
Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 7:
Simulation Event : PutC Trom : v1
Port : U1 Time : 9
History:
Event Outcome : NOTYET HANDLED
cause : PutC Trom u1 time 9
End of Simulation Event List.

Result:

The current simulation time after this query is 2. The effect of this query is to start the simulation at the time slot prior to the modification in this case at time two , with a new set of events that does not contain the deleted event and all its consequences. This will help the user in getting a different scenario of the simulation.

What if we insert an event ?

Since the user represents the environment the user is only allowed to insert environmental events. All the other events are reactions to stimulus and the user cannot insert them into the *Simulation Event List* them.

This is a sample dialogue with the user:

Enter event label: PutC

Enter trom label: u1

Enter port label: VS1

Enter occur time: 2

That is, the user wants to insert the simulation event *PutC* at time 2. If the user invokes this query after removing the event *PutD* which was scheduled at time 3 in the TROM object *u1* on port *VS1* the effect is as follows.

Since the *Simulation Event List* prior to the execution of this query is the same as the *Simulation Event List* after the previous query we will not show it again. We will only show the resulting *Simulation Event List.*

Simulation Event List after the execution of the query.

Simulation Event List:

Sim-Event 1:

Simulation Event : Next Trom : u1

Time : 0

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : idle

Assignment Vector prior to transition :

AssignmentVector :

Event Consequence :

cause : internal unconstrained event

Sim-Event 2:

Simulation Event : PutC Trom : u1

Port : VS1 Time : 2

History:

Event Outcome : NOTYET HANDLED

cause : event entered by the user

Sim-Event 3:
Simulation Event : PutC Trom : v1
Port : U1 Time : 2
History:
Event Outcome : NOTYET HANDLED
cause : PutC Trom u1 time 2
Sim-Event 4:
Simulation Event : PutD Trom : u1
Port : VS1 Time : 5
History:
Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 5:
Simulation Event : PutD Trom : v1
Port : U1 Time : 5
History:
Event Outcome : NOTYET HANDLED
cause : PutD Trom u1 time 5
Sim-Event 6:
Simulation Event : PutC Trom : u1
Port : VS1 Time : 7
History:
Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 7:
Simulation Event : PutC Trom : v1
Port : U1 Time : 7
History:
Event Outcome : NOTYET HANDLED
cause : PutC Trom u1 time 7
Sim-Event 8:
Simulation Event : PutC Trom : u1
Port : VS1 Time : 9
History:

Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 9:
Simulation Event : PutC Trom : v1
Port : U1 Time : 9
History:
Event Outcome : NOTYET HANDLED
cause : PutC Trom u1 time 9
End of Simulation Event List.

Result:

The simulation time after the execution of this query is 1. As you can see the new *Simulation Event List* contains the simulation event at the correct time, and in the correct sequence. The effect of this query is like the previous one, it helps the user to see different simulation scenarios by allowing him to modify the simulation event list.

What if we reschedule an event ?

The user can reschedule all constrained events only within the allowable timing interval. The user can reschedule unconstrained output events (environmental events). The user is not allowed to reschedule an input event; In order to do that the user has to reschedule the corresponding output event. As we have mentioned in the simulation algorithm,(appendix B), the *Simulator* will choose a random time within the allowable timing interval to schedule a constrained event. However we may want to simulate in a more controlled environment, allowing the user to test a specific scenario where the time of a simulation event is dictated by the user. The advantage of this approach is that the user will be able to test the system for a specific set of times without relying on the randomness of the simulator. We cannot allow the user to enter times that are not within the allowable timing intervals otherwise we would be violating the requirements of the simulation. By changing the time of a simulation event we may change the entire simulation. The event may be disabled, or it may enable another, thus changing completely the results of the simulation. Whenever we reschedule an event we have to rollback to the minimum of the two times: the time when it was scheduled originally and the time when it is going to be scheduled.

This is a sample dialogue:

Enter event label: SensedD

Enter trom label: v1

Enter the old occur time of the simulation event: 4

Enter the new occur time of the simulation event: 5

Enter port label: Q1

That is the user wants to modify the timing of the event *SencedD* on the TROM object *v1* from time 4 to time 5. The user can use one of the *When queries* to determine when the event was scheduled previously.

If this query is asked at time 3, with the original simulation event list shown in the previous section, a sample Simulation Event List prior to the query is:

Simulation Event List:

Sim-Event 1:

Simulation Event : Next Trom : u1

Time : 0

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : idle

Assignment Vector prior to transition :

AssignmentVector :

Event Consequence :

cause : internal unconstrained event

Sim-Event 2:

Simulation Event : PutD Trom : u1

Port : VS1 Time : 3

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : ready

Assignment Vector prior to transition :

AssignmentVector :

Event Consequence :

cause : event entered by the user

Sim-Event 3:

Simulation Event : Resume Trom : u1

Time : 3

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : place

Assignment Vector prior to transition :

AssignmentVector :

Event Consequence :

cause : internal unconstrained event

Sim-Event 4:

Simulation Event : Next Trom : u1

Time : 3

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : idle

Assignment Vector prior to transition :

AssignmentVector :

Event Consequence :

cause : internal unconstrained event

Sim-Event 5:

Simulation Event : PutD Trom : v1

Port : U1 Time : 3

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : monitor

Assignment Vector prior to transition :

AssignmentVector :

Attribute Name : inQueue

Trait Type

Trait : Trait type : PQueue Trait name : Queue

Trait value:

Queue : Size -> 0

Attribute Name : P
Trait Type
Trait : Trait type : PART Trait name : Part
Trait value:
Event Consequence :
Time Constraint:
Time Constraint: TC2 Event: SensedD
Reaction Window:
3,5
Outcome: ENABLED
cause : PutD Trom u1 time 3
Sim-Event 6:
Simulation Event : SensedD Trom : v1
Port : Q1 Time : 4
History:
Event Outcome : NOTYET HANDLED
cause : PutD Trom v1 time 3
Sim-Event 7:
Simulation Event : SensedD Trom : b1
Port : V1 Time : 4
History:
Event Outcome : NOTYET HANDLED
cause : SensedD Trom v1 time 4
Sim-Event 8:
Simulation Event : PutD Trom : u1
Port : VS1 Time : 5
History:
Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 9:
Simulation Event : PutD Trom : v1
Port : U1 Time : 5
History:
Event Outcome : NOTYET HANDLED

cause : PutD Trom u1 time 5

Sim-Event 10:

Simulation Event : PutC Trom : u1

Port : VS1 Time : 7

History:

Event Outcome : NOTYET HANDLED

cause : event entered by the user

Sim-Event 11:

Simulation Event : PutC Trom : v1

Port : U1 Time : 7

History:

Event Outcome : NOTYET HANDLED

cause : PutC Trom u1 time 7

Sim-Event 12:

Simulation Event : PutC Trom : u1

Port : VS1 Time : 9

History:

Event Outcome : NOTYET HANDLED

cause : event entered by the user

Sim-Event 13:

Simulation Event : PutC Trom : v1

Port : U1 Time : 9

History:

Event Outcome : NOTYET HANDLED

cause : PutC Trom u1 time 9

End of Simulation Event List.

The simulation event list after the query is the following:

Simulation Event List:

Sim-Event 1:

Simulation Event : Next Trom : u1

Time : 0

History:

Event Outcome : TRIGGERED TRANSITION

State prior to transition : idle

Assignment Vector prior to transition :
AssignmentVector :
Event Consequence :
cause : internal unconstrained event
Sim-Event 2:
Simulation Event : PutD Trom : u1
Port : VS1 Time : 3
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : ready
Assignment Vector prior to transition :
AssignmentVector :
Event Consequence :
cause : event entered by the user
Sim-Event 3:
Simulation Event : Resume Trom : u1
Time : 3
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : place
Assignment Vector prior to transition :
AssignmentVector :
Event Consequence :
cause : internal unconstrained event
Sim-Event 4:
Simulation Event : Next Trom : u1
Time : 3
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : idle
Assignment Vector prior to transition :
AssignmentVector :
Event Consequence :
cause : internal unconstrained event

Sim-Event 5:
Simulation Event : PutD Trom : v1
Port : U1 Time : 3
History:
Event Outcome : TRIGGERED TRANSITION
State prior to transition : monitor
Assignment Vector prior to transition :
AssignmentVector :
Attribute Name : inQueue
Trait Type
Trait : Trait type : PQueue Trait name : Queue Trait value:
Queue : Size -> 0
Attribute Name : P
Trait Type
Trait : Trait type : PART Trait name : Part
Trait value:
Event Consequence :
Time Constraint:
Time Constraint: TC2 Event: SensedD
Reaction Window:
3,5
Outcome: ENABLED
cause : PutD Trom u1 time 3
Sim-Event 6:
Simulation Event : PutD Trom : u1
Port : VS1 Time : 5
History:
Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 7:
Simulation Event : PutD Trom : v1
Port : U1 Time : 5
History:
Event Outcome : NOTYET HANDLED

cause : PutD Trom u1 time 5
Sim-Event 8:
Simulation Event : SensedD Trom : v1
Port : Q1 Time : 5
History:
Event Outcome : NOTYET HANDLED
cause : PutD Trom v1 time 3
Sim-Event 9:
Simulation Event : SensedD Trom : b1
Port : V1 Time : 5
History:
Event Outcome : NOTYET HANDLED
cause : SensedD Trom v1 time 5
Sim-Event 10:
Simulation Event : PutC Trom : u1
Port : VS1 Time : 7
History:
Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 11:
Simulation Event : PutC Trom : v1
Port : U1 Time : 7
History:
Event Outcome : NOTYET HANDLED
cause : PutC Trom u1 time 7
Sim-Event 12:
Simulation Event : PutC Trom : u1
Port : VS1 Time : 9
History:
Event Outcome : NOTYET HANDLED
cause : event entered by the user
Sim-Event 13:
Simulation Event : PutC Trom : v1
Port : U1 Time : 9

History:
Event Outcome : NOTYET HANDLED
cause : PutC Trom u1 time 9
End of Simulation Event List.
Result:
In this case the only two events that were modified are the event *SensedD* in the TROM object *v1* and its corresponding rendezvous, the event *SensedD* in the TROM object b1. The simulation time is not modified because we are modifying the timing of an event that is in the future and has not yet been handled. If the event we were rescheduling was in the past, the rollback would have changed the time of the simulation, as mentioned earlier, to the minimum of the two times, the old time and the new time.
This query is very helpful in the sense that it gives the user more control over the scenarios of the simulation without violating the requirements. If we had asked to reschedule the event to time 7 this would violate the timing constraints. This event can only occur at in the time interval 3 to five.
This is the result we get if we attempt reschedule the event to time 7:
Enter event label: SensedD
Enter trom label: v1
Enter the old occur time of the simulation event: 5
Enter the new occur time of the simulation event: 7
Enter port label: Q1
The new occur time is invalid it should be between 3 and 5

7.3.3 Using the *Reasoning System* for Validation of the Specifications

This section offers a description of the queries that are related to the static information. These queries are not related to the simulation. They can be invoked at any time.

Show all the routes between two states of a TROM object.

This is a sample dialog with the user:

Please enter trom label: r1
Please enter the initial state: S3
Please enter the final state: S3

The user wants to know all the routes between the state *S3* and the state *S3* of TROM object *r1*(the robot).

The output from the *Reasoning System* is shown below:

Route 1:
Transitions:R8;R9;R10;R4;R1;R3
Lower bound : 2
Upper bound : 8
Source state : S3
Destination state: S3
Route 2:
Transitions:R8;R9;R10;R4;R2;R3
Lower bound : 2
Upper bound : 8
Source state : S3
Destination state: S3
Route 3:
Transitions:R8;R9;R10;R5;R13;R6
Lower bound : 2
Upper bound : 6
Source state : S3
Destination state: S3
Route 4:
Transitions:R8;R9;R11;R12
Lower bound : 0
Upper bound : 2
Source state : S3
Destination state: S3
Route 5:
Transitions:R7;R9;R10;R4;R1;R3
Lower bound : 2

Upper bound : 8

Source state : S3

Destination state: S3

Route 6:

Transitions:R7;R9;R10;R4;R2;R3

Lower bound : 2

Upper bound : 8

Source state : S3

Destination state: S3

Route 7:

Transitions:R7;R9;R10;R5;R13;R6

Lower bound : 2

Upper bound : 6

Source state : S3

Destination state: S3

Route 8:

Transitions:R7;R9;R11;R12

Lower bound : 0

Upper bound : 2

Source state : S3

Destination state: S3

Result:

This is a list of all the routes, the lower and upper bound are the minimum and maximum time needed for this route not taking into consideration other TROM objects.

This a very helpful query. It allows the user to see if there are any undesirable routes. If this query is applied on all the states, this query will show the user if any state cannot be reached. This will help the user to modify his design to eliminate these errors. This query will also show the user the minimum time and the maximum time it takes to go from one state to another, using this route; This query may also help the user to modify the timing constraints.

Show a route to get to a specific state in a TROM object

This is a sample dialogue with the user:

Please enter trom label: r1

Please enter the destination state: S6

That means that we want to know a sequence of events and transitions that would lead *r1* to state *S6*. The route will show the transitions in all the related trom objects, assuming that the assertions are true.

You will note that this query checks to see if there are any possible problems in the timing constraints, and if it is the case it will show it, by saying this event may not occur due to timing constraints. In our case study the event *RecC* in the robot object may not occur. This is because the robot may still be in state *S2* when it receives this event and hence it will not be able to handle it. This flaw in the design of the Robotics System was detected by the *Reasoning System* and confirmed by the *simulator*. After a long series of runs we observed that sometimes the events *RecC* or *RecD* in the robot were handled but sometimes these events were not handled.

If the TROM object *v1* was not in the system the query would have detected that and the answer would have been that this state is no reachable.

The following result is given by the system:

Event RecC may not occur due to timing constraints conflicts

Route list:

transition : R1 lower : 0 upper : 0

transition : R2 lower : 0 upper : 0

transition : R4 lower : 0 upper : 0

transition : R1 lower : 0 upper : 0

transition : R2 lower : 0 upper : 0

Trom Object Label: u1

Lower bound : 0

Upper bound : 0

Source state : idle

Destination state: place

transition : R1 lower : 0 upper : 0

transition : R4 lower : 0 upper : 2

transition : R9 lower : 0 upper : 5
transition : R1 lower : 0 upper : 0
transition : R4 lower : 0 upper : 2
transition : R7 lower : 0 upper : 5
Trom Object Label: v1
Lower bound : 0
Upper bound : 14
Source state : monitor
Destination state: monitor
transition : R2 lower : 0 upper : 7
transition : R3 lower : 0 upper : 2
transition : R7 lower : 0 upper : 7
transition : R9 lower : 0 upper : 2
transition : R10 lower : 0 upper : 0
Trom Object Label: r1
Lower bound : 0
Upper bound : 18
Source state : S1
Destination state: S6

Result:

The state is reachable, however there may be a timing conflict in the route that may cause event *RecC* to be discarded. As you can see the timing in this query is relevant to the other TROM objects.

Chapter 8

Conclusion and Future Work

8.1 Work synthesis

This thesis is a contribution to simulated debugging and reasoning of real-time reactive systems built in TROMLAB environment. The thesis discusses the following topics:

- Chapter 2 briefly reviewed the architecture and design components of TROMLAB environment.
- In Chapters 3 and 4 we discussed the re-engineering process that we did on the *Interpreter* and the *Simulator*. We explained the need for re-engineering.
- Chapter 5 discussed a set of requirements for the *Reasoning System*.
- Chapter 6 gave a description of the design and implementation of the *Reasoning System*
- Chapter 7 gave a case study showing the results from the *Reasoning System*, which is implemented in Java, for a *Robotics Assembly Example*.

8.2 Future Work

The following are some suggested future improvements:

8.2.1 Interpreter

Parser of LSL traits should be modified to handle the complete LSL trait file, instead of the partial one which was used i.e., it should include the axioms section to the existing one. These axioms could be represented by assertion trees using JJTree. Parameterised events should be allowed to enhance the expressive power of the specifications. This will require research into the representational and behavioral aspects for parameterised events, before making changes to the parser and the *Interpreter*.

8.2.2 Simulator

1. A library consisting of the implementation of a large number of LSL trait functions could be added to the Simulator. This would allow the user to make use of different LSL traits. In the current version of the simulator only one LSL trait (Set) is implemented.

2. In current version of the Object Model support only boolean operators can be evaluated; in future, arithmetic operators should be implemented.

8.2.3 Reasoning System

1. The routes found in the last part of the *Reasoning System* do not take into consideration the truth values of the assertions. In order to do that we have to dynamically keep a trace of the values of the attributes and have to evaluate the predicates in the specifications. Some of the reasoning left to the user can be mechanised.

2. In the last query of the reasoning system we find one route to a specific state of a TROM object. This is a good reachability study. This query could be improved to find all the routes. This will give the user more scenarios to work with. However, in the presence of periodic or cyclic event structure, the number of paths (when the time is included) may be exponential.

Bibliography

[AAM96] V. S. Alagar, R. Achuthan, and D. Muthiayen. TROMLAB: A software development environment for real-time reactive systems. Submitted for publication in *ACM Transactions on Software Engineering and Methodology (First version)*, October 1996, October 1999 (Revised version).

[AAR95] R. Achuthan, V. S. Alagar, and T. Radhakrishnan. TROM - an object model for reactive system development. In *The 1995 Asian Computing Science Conference, ASIAN'95*, Thailand, December 1995.

[Ach95] R. Achuthan. *A Formal Model for Object-Oriented Development of Real-Time Reactive Systems.* PhD thesis, Department of Computer Science, Concordia University, Montréal, Canada, October 1995.

[AM98] V. S. Alagar and D. Muthiayen. Specification and verification of complex real-time reactive systems modeled in UML. Submitted for publication in *IEEE Transactions on Software Engineering (Being revised)*, July 1998.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specifications.* Springer Verlag, 1993.

[HL94] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proceedings of the 15th IEEE Real-Time Systems Symposium, RTSS'94*, pages 120–131, San Juan, Puerto Rico, December 1994.

[JKSSS90] H. Jarniven, R. Kurki-Suonio, M. Sakkinen, and K. Systa. Object-oriented specifications of reactive systems. In *Proceedings of 12th IEEE Conference on Software Engineering*, 1990.

[Mut96] D. Muthiayen. Animation and formal verification of real-time reactive systems in an object-oriented environment. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, October 1996.

[Mut98] D. Muthiayen. Real-time reactive system development – a formal approach based on UML and PVS. In *Proceedings of Doctoral Symposium held at Thirteenth IEEE International Conference on Automated Software Engineering, ASE98*, Honolulu, Hawaii, October 1998.

[MA99] Muthiayen, D. and Alagar, V.S. Mechanized Verification of Real-Time Reactive Systems in an Object-Oriented Framework Submitted to IEEE Software Transactions on Software Engineering, 1999.

[Nag99] R. Nagarajan. Vista - a visual interface for software reuse in TROMLAB environment. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, April 1999.

[Oan99] Oana, P. Rose-GRC translator:Mapping UML visual models onto formal specifications. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, September 1999.

[Obj97] Overcoming the crisis in real-time software development. Technical report, ObjecTime Limited, 1997.

[Obj98] Uml for real-time overview. Technical report, ObjecTime Limited, 1998.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: a prototype verification system. In *Proceedings of 11th International Conference on Automated Deduction, CADE*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, New York, 1992. Springer Verlag.

[Pom99] F. Pompeo. A formal verification assistant for TROMLAB environment. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, March 1999. Under preparation.

[Rat97] Rational Software Corporation. *UML Notation Guide, Version 1.1*, September 1997.

[Rat98a] Rational Software Corporation. *Rational Rose 98 Enterprise Edition Rose Extensibility Interface*, February 1998.

[Rat98b] Rational Software Corporation. *Rational Rose 98 Using Rose*, February 1998.

[RS98] J. Rumbaugh and B. Selic. Using uml for modeling complex real-time systems. Technical report, March 1998.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[Shin92] R. Shinghal. Formal Concepts in Artificial Intelligence. Chapman and Hall

[Sri99] V. Srinivasan. Graphical user interface for TROMLAB environment. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, March 1999. Under preparation.

[Tao96] H. Tao. Static analyzer: A design tool for TROM. Master's thesis. Department of Computer Science, Concordia University, Montréal, Canada. August 1996.

Appendix A

GRC, and SCS Grammar

GRC	::=	<class> <events> <states> <attributes> <traits> <att_funcs> <tran_specs> <time_constraints> end

Table 2: Grammar for generic reactive class specification

In the grammar, a class (see Table 3) is described by the keyword Class, followed by a string denoting the class name, followed by a list of port types in square brackets . The list of port types is composed of one or several port type names, represented as strings starting with the symbol @ and separated by a comma.

class	::=	Class <class_name> [<port_types>] NL
port_types	::=	<port_type_name> | <port_type_name>, <port_types>
class_name	::=	String
port_type_name	::=	@String

Table 3: Grammar for generic reactive class title

Events (see Table 4) are introduced by the keyword Events, followed by the list of events. The list of events can contain one or several events, separated by comma. Each event can be an internal event, an input event or an output event. Internal events are represented by a string for the event name. Input events are represented by a string as event name, followed by the character ? and the string for the port type at which the event occurs. Output events are represented by a string as event name, followed by the character ! and the string for the port type at which the event occurs.

States (see Table 5) are introduced by the keyword States, followed by the state set. The state set is comprised of the initial state, followed by a list of one or several states, separated by comma. A state is represented by a string for the name. If the state is complex, the name is followed by its substates, represented as a state set, within curly braces.

events	::=	**Events:** <event_list> **NL**
event_list	::=	<event> | <event>, <event_list>
event	::=	<inputevent> | <outputevent> | <interevent>
inputevent	::=	<event_name> ? <port_type_name>
outputevent	::=	<event_name> ! <port_type_name>
interevent	::=	<event_name>
event_name	::=	String
port_type_name	::=	@String

Table 4: Grammar for events

states	::=	**States:** <state_set> NL
state_set	::=	*<state>, <state_list>
state_list	::=	<state> | <state>, <state_list>
state	::=	<state_name> | <state_name><state_set>
state_name	::=	String

Table 5: Grammar for states

Attributes (see Table 6) are introduced by the keyword **Attributes**, followed by the list of attributes. The list of attributes is comprised of one or several attributes, separated by a semi-colon. Attributes of type port type are represented by a string for the attribute name, followed by colon and by the port type name, which starts with the character @. Attributes of type data type are represented by a string for the attribute name, followed by a colon and by the LSL trait type name.

LSL traits (see Table 7) are introduced by the keyword **Traits**, followed by a list of traits. The list of traits is comprised of one or several traits. A trait is represented as a string for the trait name, followed in square brackets by the argument list and

attributes	::=	Attributes: <att_list>**NL**
att_list	::=	<attribute> | <attribute>;<att_list>
attribute	::=	<att_name> : <port_type_name> | <att_name> : <trait_type_name> | <att_name> : **Integer** | <att_name> : **Boolean**
att_name	::=	String
trait_type_name	::=	String
port_type_name	::=	@String

Table 6: Grammar for attributes

traits	::=	Traits: <trait_list> NL
trait_list	::=	<trait> | <trait>, <trait_list>
trait	::=	<trait_name>[<arg_list>,<trait_type_name>] | <trait_name>[<trait_type_name>]
arg_list	::=	<arg> | <arg>, <arg_list>
arg	::=	<trait_type_name> | <port_type_name>
trait_name	::=	String
trait_type_name	::=	String
port_type_name	::=	@String

Table 7: Grammar for LSL traits

att_funcs	::=	Attribute–Function: <att_func_list>
att_func_list	::=	<att_func>; | <att_func>;<att_func_list>
att_func	::=	<state_name> → <att_list> NL
att_list	::=	<att_name> | <att_name>,<att_list> | empty
att_name	::=	String
state_name	::=	String

Table 8: Grammar for attribute functions

the trait type name. The argument list is comprised of one or several arguments. An argument is either a trait type name or a port type name starting with the character @.

The attribute function (see Table 8) is introduced by the keyword Attribute-Function, followed by a list of attribute function applications. The list of attribute function applications has one or several attribute function applications, separated by a semi-colon. Each attribute function application is comprised of the state name as a string, followed by the keyword →, followed by an attribute list, between curly braces. An attribute list is comprised of zero or several attribute names, separated by a comma.

Transition specifications (see Table 9) are introduced by the keyword Transition-Specifications, followed by the list of transition specifications, separated by semi-colons and new lines. The list of transition specifications is composed of one or several transition specifications, separated by new lines. A transition specification consists of a name, followed by a colon, one or several state pairs, separated by semi-colons, a triggering event, an assertion, the implication operator → and another assertion. A state pair consists of two state names, in brackets, separated by a comma. The triggering event is an event name followed in brackets by an assertion. An assertion is either a

tran_specs	::=	**Transition–Specifications:** NL <tran_spec_list>
tran_spec_list	::=	<tran_spec> NL | <tran_spec> NL <tran_spec_list>
tran_spec	::=	<tran_spec_name>: <state_pairs> <trig_event> <assertion> → <assertion>;
state_pairs	::=	<state_pair>; | <state_pair>; <state_pairs>;
state_pair	::=	(<state_name>,<state_name>)
trig_event	::=	<event_name>(<assertion>)
assertion	::=	<simple_exp> | <simple_exp> <b_op> <simple_exp>
b_op	::=	= | ≠ | > | ≥ | < | ≤
simple_exp	::=	<term> | <term> <OR> <term>
term	::=	<factor> | <factor> <AND> <factor>
factor	::=	<NOT> <factor> | pid | <att_name/ > | <att_name> | true | false | <LSL_term> | (<assertion>)
LSL_term	::=	<LSL_func_name>(<arg_list>)
arg_list	::=	<arg>|<arg>,<arg_list>
arg	::=	pid | <att_name> | <LSL_term>
att_name/	::=	String
att_name	::=	String
state_name	::=	String
event_name	::=	String
LSL_func_name	::=	String
OR	::=	|
AND	::=	&
NOT	::=	!

Table 9: Grammar for transition specifications

simple expression or two simple expressions with a binary operator between them. A binary operator is one of: $=$, $\neq$, $<$, $\leq$, $>$, $\geq$. A simple expression is either a term or two terms with the | logical operator. A term is either a factor, or two factors with the & logical operator. A factor can be the logical operator ! followed by a factor, or the reserved variable **pid**, or a primed attribute, an attribute, logical expressions *true* or *false*, an LSL term or an assertion in brackets. An LSL term consists of a LSL function name, followed by an argument list in brackets. An argument list is composed of one or several arguments. An argument is either the reserved variable pid, or an attribute name or an LSL term. A primed attribute is an attribute (from the attribute function) followed by the character *'*.

Time constraints (see Table 10) are introduced by the keyword **Time-Constraints**, followed by one or several constraints, separated by semi-colons and new lines. A

time_constraints	::=	Time–Constraints: NL <constraints>
constraints	::=	<constraint>; NL | <constraint> ; NL <constraints>
constraint	::=	<time_cons_name>: <tran_spec_name>, <event_name>, <min_type><min>,<max><max_type>,<states>
states	::=	<state_name>|<state_name>,<states> | empty
state_name	::=	String
time_cons_name	::=	String
tran_spec_name	::=	String
event_name	::=	String
min	::=	NAT
max	::=	NAT
min_type	::=	(| [
max_type	::=	) |]

Table 10: Grammar for time constraints

SCS	::=	SCS <scs_name> NL <include> <instantiates> <configure> end
scs_name	::=	String

Table 11: Grammar for subsystem configuration

constraint has a name followed by colon and the name of the constraining transition specification, the name of the constrained event, the lower and upper bounds, and a list of disabling states. The lower and upper bounds are preceded and followed, respectively, by the open or closed interval indicators. The list of disabling states is comprised of zero, one or several state names, separated by a comma.

The configuration specification should respect the following grammar, introduced in [Tao96].

A subsystem configuration specification (see Table 11) is introduced by the keyword SCS, followed by its name as a string, a new line and the following sections: **Includes**, **Instantiates**, **Configure**, all followed by the keyword **end**.

The include section (see Table 12) is introduced by the keyword **Includes**, followed by a list of subsystem names and a new line. The list of subsystem names is composed of one or several subsystem names, separated by a semi-colon.

The instantiates section (see Table 13) is introduced by the keyword **Instantiate**, followed by an instance list and a new line. An instance list is composed of one or several instances. An instance consists of an object name, followed by two colons, a generic

include	::=	**Includes:** <scs_name_list> NL
scs_name_list	::=	<scs_name>; | <scs_name_list>
scs_name	::=	String

Table 12: Grammar for include section

instantiates	::=	**Instantiate:** <inst_list> NL
inst_list	::=	<instantiate>; NL | <instantiate>; NL <inst_list>
instantiate	::=	<obj_name>::<grc_name>[<port_card_list>]
port_card_list	::=	<port_card>|<port_card>,<port_card_list>
port_card	::=	<port_type_name>:<cardinality>
obj_name	::=	String
port_type_name	::=	@String
grc_name	::=	String
cardinality	::=	NAT

Table 13: Grammar for instantiate section

class name and, in square brackets, by a port cardinality list. The port cardinality list is composed of one or several port cardinalities. A port cardinality is represented by a port type name, followed by a colon and a natural number for the cardinality.

The configure section (see Table 14) is introduced by the keyword **Configure**, followed by the object port list. The object port list is composed by one or several object port links, separated by a semi-colon. An object port link is composed of an object name, followed by a period, a port name starting with character @ and its port type, the composition operator $\leftrightarrow$, another object name, followed by a period, and a port name starting with character @ and its port type.

configure	::=	**Configure:** <obj_port_list>
obj_port_list	::=	<obj_port_link>; NL | <obj_port_link>; NL <obj_port_list>;
obj_port_link	::=	<obj_name>.<port_name>:<port_type_name> $\leftrightarrow$ <obj_name>.<port_name>:<port_type_name>
obj_name	::=	String
port_name	::=	@String
port_type_name	::=	@String

Table 14: Grammar for configure section

Appendix B

Simulation Algorithm

```
begin /*simulation algorithm */
    process TROM classes to be used in simulation
    instantiate Subsystem s
    instantiate subsystems included in Subsystem s
    instantiate TROM objects included in Subsytem s
    instantiate TROM objects for each Subsytem
    initialize CurrentState and Assignement vector for each TROM object
    configure port links for each Subsytem
    initialize simulation clock
    schedule unconstrained internal events from initial state for each TROM
    object
    for all SimulationEvents se in SimulationEventList sel
    begin /* at this stage simulation clock can be frozen and debugger can be
    activated */
            while simulation clock < occur time of se
            begin
                    increment SimulationClock /* using machine clock */
            end
            while exists SimulationEvent se and
            SimulationClock == Occur time of se
            begin /* handel simulation event se */
                    get TROM object trom accepting SimulationEvent se from
                    Subsystem s
                    get TransitionSpec ts triggered by SimulationEvent se
                    /* update history of SimulationEvent se */
                    save CurrentState of TROM object trom in EventHistory of se
                    save Assignment Vector of TROM object trom in
                    EventHistory of se
                    /* update status of TROM object trom */
```

```
change CurrentState of TROM object trom to DestinationState
of TransitionSpec ts
change AssignmentVector of TROM object trom according to
post condition of ts
/* handel transition specified by transition ts */
for all TimeConstraint tc in list of TimeConstraints for
TROM object trom
begin
        if constrained event of TimeConstraint tc == label
        of SimulationEvent se
        begin
                for each ReactionWindow rw in
                reaction subvector associated with tc
                begin
                        if SimulationEvent se occurs
                        within ReactionWindow rw
                        begin /* fire reaction according to
                        TimeConstraint tc */
                                Remove ReactionWindow rw from
                                reaction subvector associated
                                with tc
                                insert ReactionHistory rh in
                                EventHistory of se
                                according to rw
                        end
                end
        end
        if current state of TROM object trom is in
        set of disabling states tc
        begin /* disable reaction according to
        TimeConstraint tc */
                for all Reaction Windows rw in
                reaction subvector associated with tc
                begin
```

```
                    remove ReactionWindow rw from
                    reaction subvector ass.whith tc
                    insert ReactionHistory rh in
                    EventHistory of se according to rw
                    unschedule disabled SimulationEvent
                    in SimulationEventList sel
                    if constrained event of
                    TimeConstraint tc is an output event
                    begin
                        remove disabled SimulationEvent
                        scheduled for syncronization
                    end
                end
            end
            if label of TransitionSpec ts == transition label of
            TimeConstraint tc
            begin /* enable reaction according to
            TimeConstraint tc */
                insert new ReactionWindow rw in
                reaction subvector associated whith tc
                insert ReactionHistory of se according to rw
                /* shedule new SimulationEvent */
                insert new SimulationEvent se2
                in SimulationEventList sel
                using lru port of port type of
                constrained event tc and
                random time within
                ReactionWindow rw
            end
        end
        schedule unconstrained internal event from current state for
        TROM object trom
        if constrained event of TimeConstraint tc is an output event
        begin /* identify linked TROM object for syncronization */
```

```
                        get PortLink pl from subsytem s linking the two
                        TROM objects
                        /* shedule new SimulationEvent */
                        insert new SimulationEvent se3 in
                        SimulationEventList sel
                        using port pl for sycronization
                    end
                    get next SimulationEvent se from simulationEventList sel
                end
            end
        end /* simulation algorithm */
```