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®

GRAPHICAL USER INTERFACE FOR TROMLAB ENVIRONMENT

VangalurSrinivasan Vagula Bhaskaran

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

Graphical User Interface for TROMLAB environment

VangalurSrinivasan Vagula Bhaskaran

Real-time reactive systems are characterized by their continuous interaction with their environment through stimulus-response behaviour. The safety-critical nature of their domain and their inherent complexity advocate the use of formal methods in the software developement process. An effective user interface for real-time reactive systems developement environment would hide the complexity of the formalism in the design, and promote ease of use.

This thesis addresses the design and implementation of a *Graphical User Interface (GUI)* which provides precise interaction points for the user-centered tasks of TROMLAB, a rigorous real-time reactive systems development environment being built in the Department of Computer Science, Concordia University.

The scope of this thesis is two-fold. We first reengineer the initial TROMLAB design, which is a prerequisite to *GUI* developement. Reengineering was necessary due to the need for a flexible design with abilities to absorb changing requirements with minimal changes to the design, and usability of the entire TROMLAB environment. We then build an integrated *GUI* to interact with TROMLAB environment.

Acknowledgments

I would like to take this opportunity to extend my gratitude to **Dr.Alagar**, my thesis supervisor and mentor, for the motivation, technical support and financial support he has given through my studies. This work would not have been possible without his consistent guidance and encouragement.

I gratefully acknowledge the graduate awards office for providing me international student fee remission award for the years 1997-98, and 1998-99.

I wish to thank my parents for encouraging me to pursue graduate education and for their moral and financial support. Their love, and support sustained my spirits during the long hours I spent working on this thesis.

I wish to thank **Stan Swiercz** for his valuable suggestions, and technical support during the developement process.

I would like to thank **Darma** for the useful discussions, and suggestions. Many thanks goes to my colleague **Haidar** for his support during the developement process, and for providing a lively work place.

Contents

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Real Time Reactive Systems 1

1.2 Scope of the Thesis 3

2 TROMLAB Environment - a brief review of initial design 5

2.1 TROMLAB Formalism 6

2.1.1 Data Abstraction Tier 8

2.1.2 TROM Tier 8

2.1.3 Subsystem Specification Tier 9

2.2 Syntax and Semantics 9

2.3 TROMLAB Components 14

2.3.1 The *Interpreter* 15

2.3.2 The *Simulator* 17

3 Reengineering TROMLAB 19

3.1 Usability Analysis of TROMLAB 19

3.1.1 Need for improving the usability of TROMLAB 20

3.1.2 Need for reengineering TROMLAB 20

3.1.3 The revised TROMLAB 21

3.2 Improvements 23

3.2.1 *Interpreter* 24

3.2.2 *Simulator* 27

3.3 Revised User Requirements 28
3.3.1 Requirements of the *Interpreter GUI* 28
3.3.2 Requirements of *Simulator GUI* 28
3.3.3 Requirements of *Reasoning system GUI* 30
3.3.4 Requirements of the *GUI* for other components of TROMLAB 31

4 Design and Implementation of the Modified *Interpreter* and *Simulator* 35
4.1 Class diagrams 35
4.1.1 *Interpreter* 35
4.1.2 *Simulator* 40
4.2 Language of choice 42
4.2.1 JavaCC 42
4.2.2 JJTree 43
4.3 Implementation 43
4.3.1 *Interpreter* 43
4.3.2 *Simulator* 48
4.3.3 Interfacing with the *Simulator* 49

5 Graphical User Interface : Design and Implementation 51
5.1 *GUI* design issues 51
5.2 Detailed Design of *GUI* 52
5.2.1 *Interpreter GUI* 53
5.2.2 *Simulator GUI* 55
5.3 *GUI* Implementation 56
5.3.1 Snapshot of *Interpreter GUI* 57
5.3.2 Snapshot of *Simulator GUI* 59

6 Case Study : Robotics Assembly example 65
6.1 Introduction 65
6.2 Problem Description 65
6.2.1 Informal Problem Description 65
6.2.2 Class Diagram for *Robotics Assembly* 68
6.2.3 Formal Problem Description 69
6.3 *GUI* for *Robotic Assembly* 78
6.3.1 *Interpreter GUI* 78

6.3.2 *Simulator GUI* 79

7 Conclusion and Future Work **89**

7.1 Future work 89

Bibliography **91**

Appendix A **94**

Appendix B **100**

Appendix C **104**

.1 *Interpreter GUI* 104

.2 *Simulator GUI* 107

List of Figures

1 Three tier 7
2 Set trait 8
3 Train class specifications 10
4 Gate class specifications 10
5 Controller class specifications 11
6 SCS 11
7 Architecture of interpreter 16
8 Architecture of simulation tool 18
9 Future TROMLAB environment 23
10 AST Structure 26
11 *Interpreter* Class diagram (Old) 36
12 *Interpreter* Class diagram - Detailed (Old) 36
13 *Interpreter* Class diagram - Detailed (Old) 37
14 *Interpreter* Class diagram (New) 38
15 *Interpreter* Class diagram - SCS (New) 39
16 *Interpreter* Class diagram - TROMclass (New) 39
17 *Simulator* Class diagram - TROM class diagram(New) 40
18 *Simulator* Class diagram - Simulation Event Object model (New) . . 41
19 *Simulator* Class diagram - Subsystem Object model (New) 41
20 Simulation event list 47
21 Use Case diagram for *Interpreter GUI* 53
22 Object diagram for *Interpreter GUI* 54
23 Use Case diagram for *Simulator GUI* 55
24 Object diagram for *Simulator GUI* 56
25 Window of *Interpreter GUI* 58
26 Window of *Simulator GUI* 60

27 Window of *Simulator - Debugger GUI* 61
28 Window of *Simulator - Query Handler GUI* 62
29 Window of *Simulator - Trace Analyser GUI* 63
30 Window of *Simulator - Reasoning System GUI* 64
31 Robotics System Class diagram 69
32 User TROM class - Textual representation 70
33 User TROM class - State machine representation 70
34 User TROM class - UML model 71
35 Vision system TROM class - Textual representation 71
36 Vision system TROM class - State machine representation 72
37 Vision system TROM class - UML model 72
38 Belt TROM class - Textual representation 73
39 Belt TROM class - State machine representation 73
40 Belt TROM class - UML model 73
41 Robot TROM class - Textual representation 74
42 Robot TROM class - State machine representation 75
43 Robot TROM class - UML model 75
44 SCS - Textual representation 76
45 SCS - UML model 76
46 Sample Simulation Event List 77
47 Part LSL Trait 77
48 Queue LSL Trait 78
49 Stack LSL Trait 78
50 Window of *Interpreter GUI* 79
51 Window of *Simulator GUI* 84
52 Window of *Simulator - Debugger GUI* 85
53 Window of *Simulator - Query Handler GUI* 86
54 Window of *Simulator - Trace Analyser GUI* 87
55 Window of *Simulator - Reasoning System GUI* 88

List of Tables

1 States of Robot Manipulator. 67
2 Grammar for generic reactive class specification 94
3 Grammar for generic reactive class title 94
4 Grammar for events 95
5 Grammar for states 95
6 Grammar for attributes 95
7 Grammar for LSL traits 96
8 Grammar for attribute functions 96
9 Grammar for transition specifications 97
10 Grammar for time constraints 98
11 Grammar for subsystem configuration 98
12 Grammar for include section 99
13 Grammar for instantiate section 99
14 Grammar for configure section 99

Chapter 1

Introduction

Research in real-time reactive systems revolve around four important topics:

- languages and methods for specification and design,
- formal techniques for verification and validation,
- development of tools for applying formalism with sufficient rigor, and
- effective user interface for hiding complexity and promoting ease of use.

This thesis addresses the design and implementation of a *Graphical User Interface(GUI)* which provides precise interaction points for the user-centered tasks 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 of 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 :

- *size:* 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 visual languages for architectural descriptions,
- mechanical generation of formal specification from visual models,
- design theories for refinement,

- process model support for iterative design, animated analysis, and design-time debugging,
- validation and reasoning based on an animation of the design,
- formal verification assistant based on PVS,
- 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

The primary goal of TROMLAB environment is to provide a framework for a rigorous development and analysis of real-time reactive systems. The application developer, who is normally an expert in the application domain, is facilitated to focus on the modeling and analysis of the problem without being burdened by formal notations. This should enhance the quality of the software as well as human productivity. Another goal is to reduce the complexity of understanding the results and behaviour of the system through the introduction of easy to use and easy to learn task-oriented descriptions in *GUI*. Strongly motivated by these two goals, TROMLAB designers have included visual representations and tools integrating the formalism and rigorous analyses to their pictorial counterparts. Class diagrams, state machine models of reactive objects, and system configuration diagrams can be constructed through the *GUI*. For composing large system configuration specifications, *GUI* is quite helpful in breaking the complexity barrier - concrete graphical representations are useful to communicate complex system architecture, interactions of its components, as well as the intuitive understanding of the system under development. There is some support to this claim as evidenced by the popularity of Statemate, and ObjecTime tools [Obj97]. The look and feel of *GUI* is influenced by these tools.

The user interface has been conceived and planned to provide interaction points to *Interpreter*, *Simulator*, *Reasoning System*, *Verifier*, and *Browser* components of TROMLAB environment. A major goal in *GUI* design is to ensure that application developer needs to know only these interaction points; it is not necessary to know

the formalism underlying the pictorial counterparts. The *Interpreter* [Tao96] and the *Simulator* [Mut96] were both simultaneously designed and implemented in C++. The next stage of TROMLAB evolution included the planning and design of the *Browser* [Nag99], the *Reasoning System* [Hai99], and the *GUI*. During this stage, the shortcomings and the inadequacies in the design and implementation of the *Interpreter* were brought to light. The need to remedy these defects and the lack of tools to design *GUI* for C++ software lead to a complete reengineering of both the *Interpreter*, and the *Simulator*.

Reengineering was directed by user feedback, and predicting the involvement of future users. Test team, verification team, and implementation team are the three important categories of future users. During the last three years the initial designers, current and future users held several brainstorming sessions, group discussions, and walkthroughs to determine the usability goals of TROMLAB. They have influenced the current *GUI* design. Interacting through *GUI*, it is easy to learn TROMLAB features and is easy to use TROMLAB features. The two contribution of this thesis are:

1. reengineering TROMLAB, which was a prerequisitive to *GUI* development, and
2. *GUI*, an integrated graphical user interface to interact with TROMLAB environment.

After a brief review of TROMLAB environment in Chapter 2, we discuss a rationale for reengineering the initial design of TROMLAB, and come up with a revised set of user requirements in Chapter 3. A complete discussion of the new design and Java implementations of the *Interpreter* and the *Simulator* appear in Chapter 4. Having thus set the stage for the proper context in which a *GUI* can be implemented, we give a set of VDM specification for describing the tasks of *GUI*, class diagrams and a detailed design of the *GUI* in Chapter 5. In Chapter 6 we illustrate *GUI* features and *GUI* interaction points with other TROMLAB components for a *Robotic Assembly* example. Chapter 7 concludes the thesis with a summary of the contributions and a list of future enhancements.

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* [Tao96] and *Animator* [Mut96] were the first components to be built. Recently, a *Browser* [Nag99] 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 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. Timed Reactive Object Model(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.

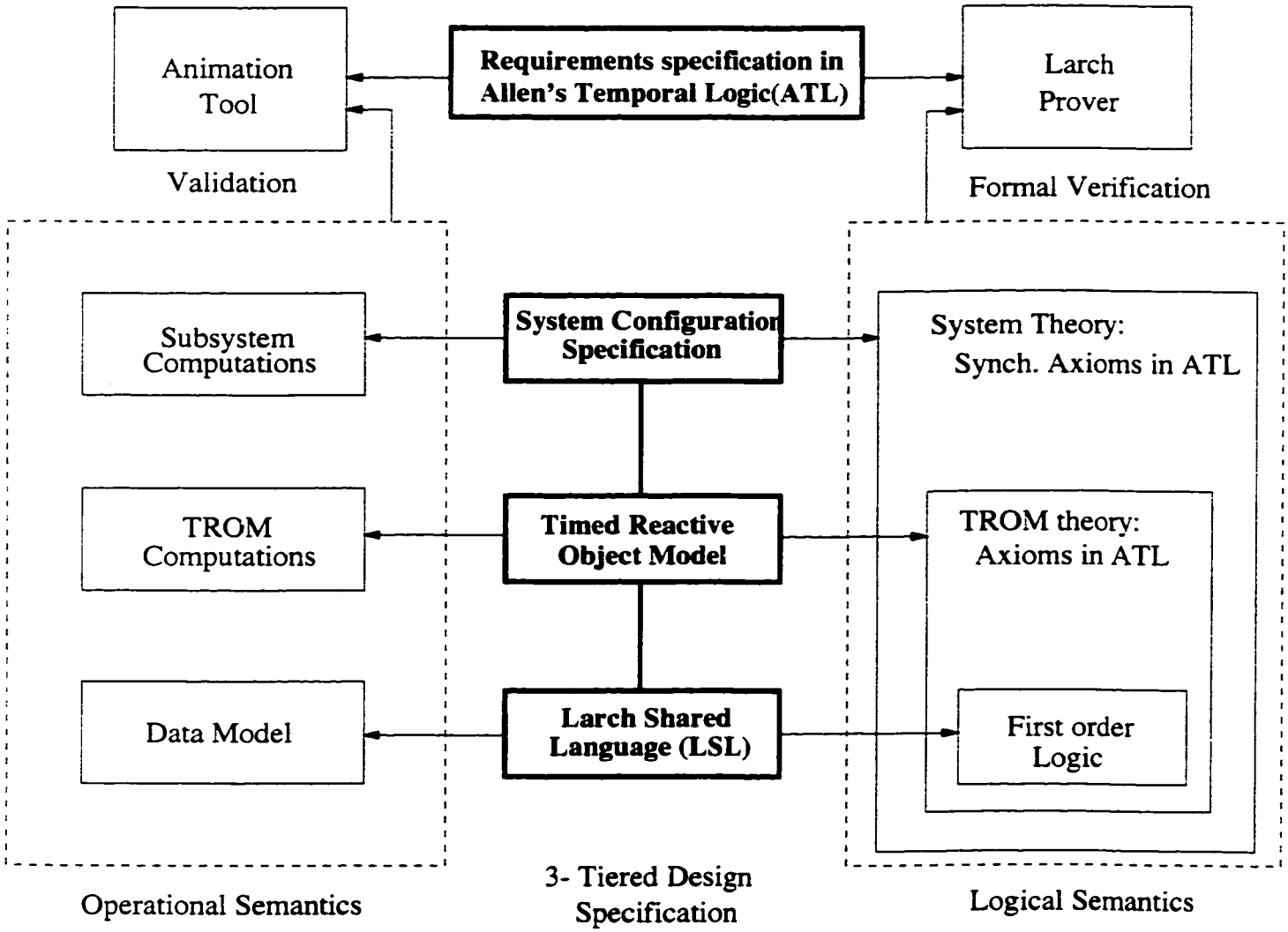


Figure 1: Three tier

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. The following LSL trait for set data type is shown in the 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 interact only through the port linking them as defined in SCS. Only compatible ports can be linked, such that an 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 this 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 of 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-condition 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-constrained 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 values 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 functionalities 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 analysis:* 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.

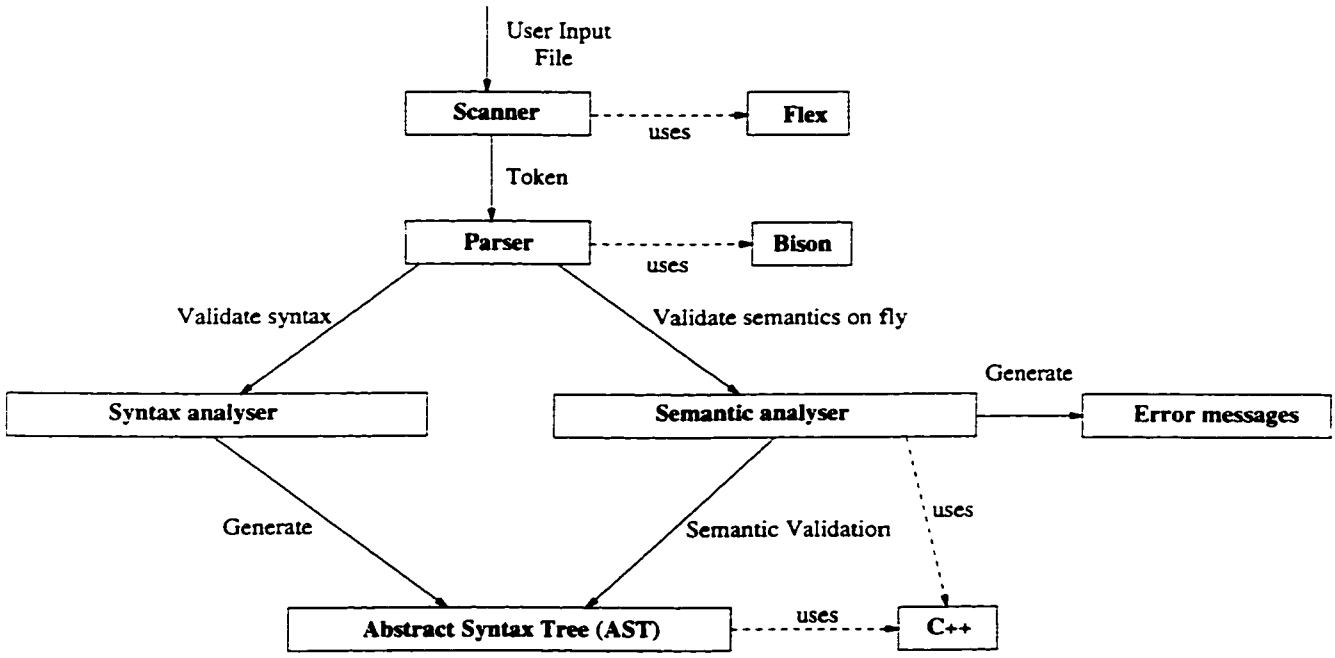


Figure 7: Architecture of interpreter

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 the next Chapter.

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 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.

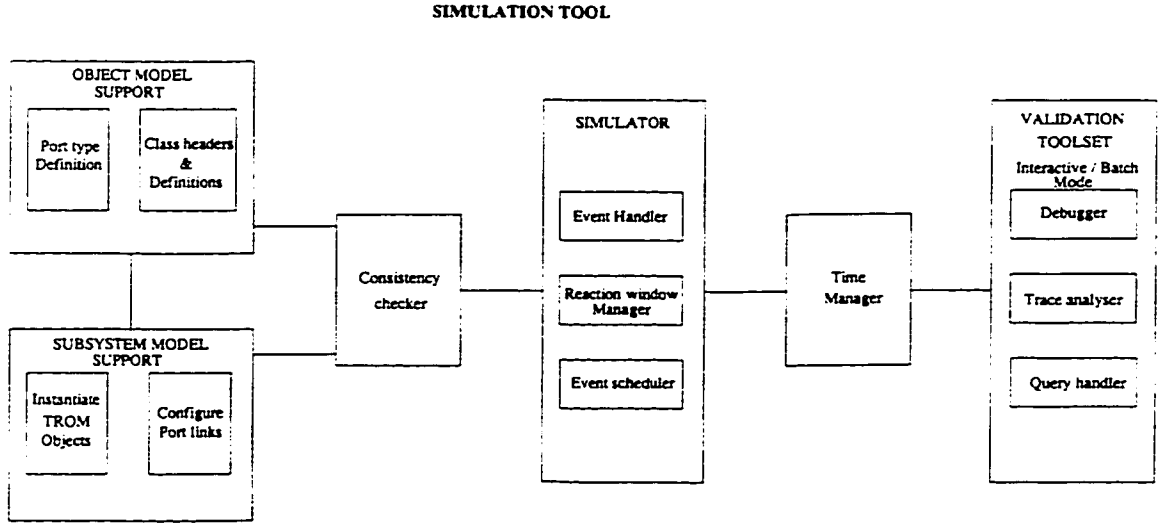


Figure 8: Architecture of simulation tool

Chapter 3

Reengineering TROMLAB

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 reengineer and rebuild several of the system components. Reengineering is part of the system maintenance process that helps to improve the performance, adapt to the changing environment.

The initial TROMLAB design is an example of a design which cannot be adapted to a graphical user interface front-end. However, a *GUI* has been recognised as an important requirement for the usability of the entire system. This is the major reason that a reengineering of the initial design was undertaken.

3.1 Usability Analysis of TROMLAB

In this section we identify the rationale for improving the usability of the TROMLAB environment, and thus state the needs for reengineering. Many of the inadequacies of TROMLAB surfaced when new components such as *Reasoning System* to work with the already existing components, such as *Interpreter* and *Simulator* were investigated for their design. Clearly, maintenance, enhancement, and support replacements of TROMLAB required a deeper understanding of the initial designs. The initial designers were not the next set of users of the system. So, the new users and software developers of TROMLAB needed to expend many resources to examine and learn about the system. Due to the lack of any reverse- engineering tools, the team had to develop their own techniques to understand the system so that appropriate changes

and extensions could be made.

3.1.1 Need for improving the usability of TROMLAB

The three important goals of usability are *learnability*, *throughput*, and *flexibility*. Based on the above goals of usability we would like to list the reasons for improvement of the usability of the initial TROMLAB environment. The reasons are as follows:

- Users did not have a mechanism to design formal specification files, in the process of designing a system.
- Absence of interactive mechanisms, such as choosing an existing formal specification file to design a system, and debugging when the simulation is running.
- Absence of flexible mechanisms which will allow user to explore, and expand the TROMLAB environment.
- Absence of different screen layouts for injecting different set of tasks.
- Absence of multiple sources on the screen to carry out a task at the same time.
- Absence of visual models to aid user in understanding the simulation of the system.

Due to the above reasons the time and effort required by an user to learn, and use the TROMLAB environment remained high. This motivated us to provide the user's of TROMLAB a more effective, interactive, flexible, visual environment. Thus, the overriding objective was to provide a user-centered design of *GUI* for TROMLAB, which is easy to learn and use, and remain safe and effective while facilitating the activities of users.

3.1.2 Need for reengineering TROMLAB

Having established that the usability has to be improved, in order to achieve it we should alter, and restructure the TROMLAB with respect to new requirements not met by the original system. Reengineering generally includes some form of reverse engineering followed by some form of forward engineering or restructuring. Reverse engineering stage of TROMLAB required the identification of the system's components

and their interrelationships, and creating representations for them in another form at a slightly higher level of abstraction. During this phase, no change to TROMLAB were made - it was only a process of critical examination. It revealed that the design of the *Interpreter* was severly restrictive:

- incremental specification and design development was not possible;
- internal representations were consistent with OO themes such as modularity, and encapsulation;
- it cannot be integrated with *GUI*

We established the goals of reengineering as follows:

1. Modify the internal mechanisms, such as data structures and programs without changing the functionality (system capabilities perceived by the user).
2. Modify the language of implementation so as to link *GUI* with all the components of TROMLAB.
3. Provide visual modeling and analysis capabilities.
4. Cope with complexity of evolution.
5. Integrate graphical and textual views through *GUI*.
6. Provide sufficient documentation and help facility.
7. Facilitate reuse.

Based on these decisions, the reengineering phase was started in which both *Interpreter* and *Simulator* were modified, *Reasoning System*, *GUI*, *Browser*, *UML Rose Interface* and *Verifier* were built.

3.1.3 The revised TROMLAB

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 important criteria for the design of TROMLAB namely *scalability*, *portability*, and *flexibility*.

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. *UML-RT support:* This tool is the front-end for visually composing reactive system specifications using UML-RT 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 to model real-time reactive systems in Rose. The UML-RT support [Oan99], extracts the information from these models and generates formal specifications in the syntax defined in TROM methodology.

5. *Reasoning system:* This tool, being built now [Hai99], 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*, whose design is given in later chapters, 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 to 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.

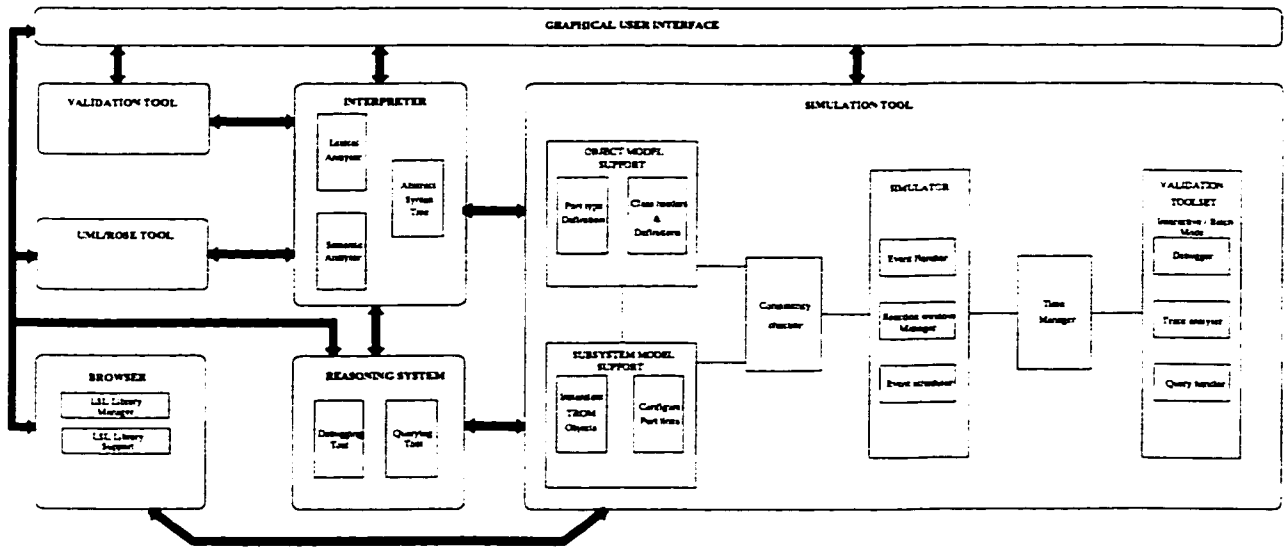


Figure 9: Future TROMLAB environment

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 reengineered components as well as the yet to be implemented components of the verifier.

3.2 Improvements

It is the *Interpreter* that required a totally new design and implementation as a result of reengineering of TROMLAB environment. 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 recompilation 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 LSL 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 object oriented 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 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.

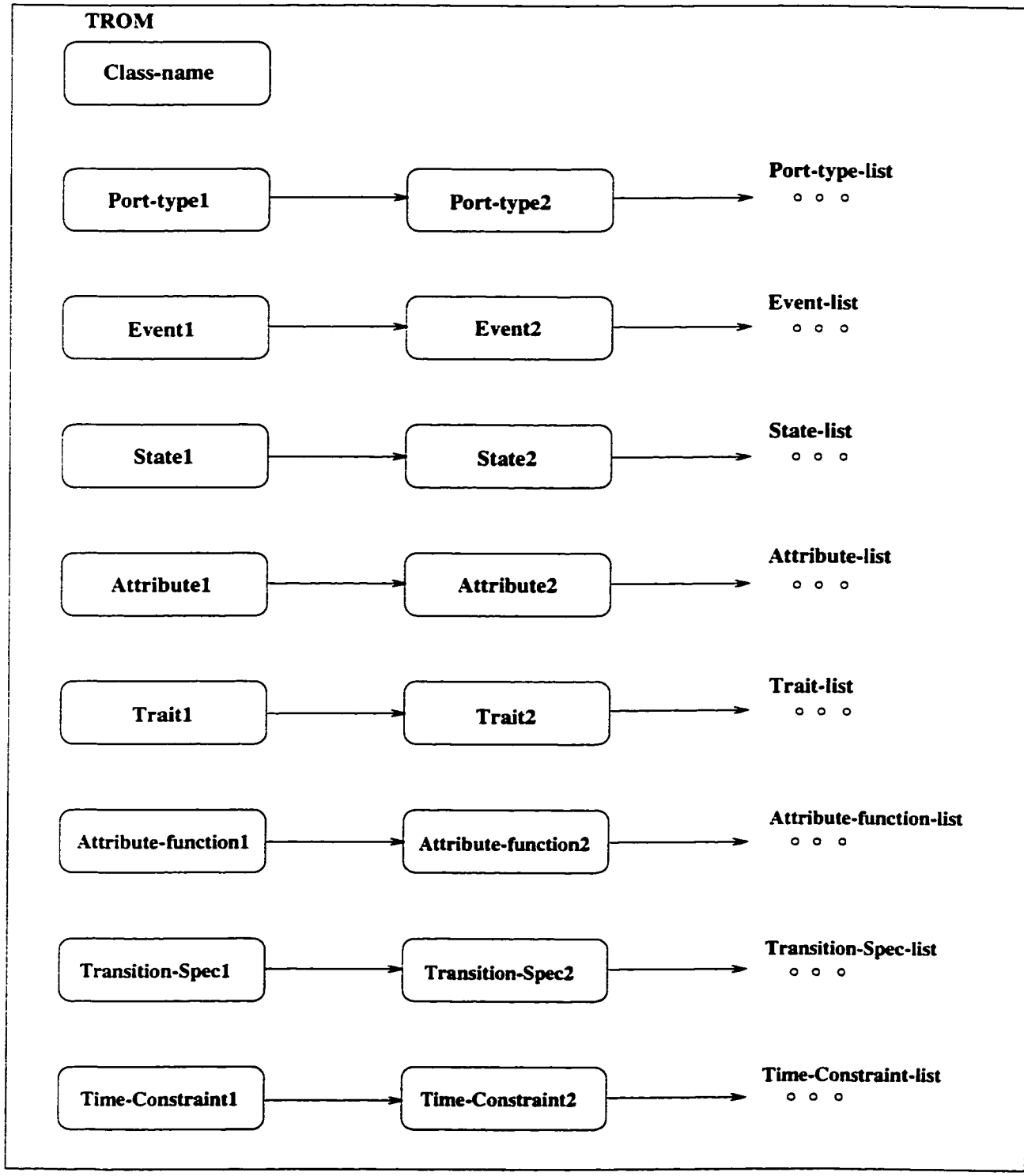


Figure 10: AST Structure

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 [Hai99]. 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 *Event Scheduler* 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 simulation of a variety of reactive system models, we added several other commonly used LSL traits, such as *Stack*, and *Queue*.

3.3 Revised User Requirements

In this section we identify the revised user requirements for the *GUI*. This section is divided into three subsections namely, requirements of the *Interpreter GUI*, requirements of the *Simulator GUI*, requirements of the *Reasoning system GUI*, and requirements of the *GUI* for other components of TROMLAB.

3.3.1 Requirements of the *Interpreter GUI*

The *GUI* should provide the following mechanisms to the user in order to use the *Interpreter*.

- A mechanism to design new TROM class specifications, open an existing TROM class specifications, and save them.
- A mechanism to design new Subsystem Configuration Specifications, open an existing Subsystem Configuration Specifications, and save them.
- A mechanism to design the Initial Simulation Event List, open an existing Initial Simulation Event List, and save them.
- A mechanism to invoke the corresponding parsers for the formal specification files.
- A mechanism to display appropriate messages at end of syntax checking.
- A mechanism to invoke semantic analysis once the user had successively parsed the formal specification files.

3.3.2 Requirements of *Simulator GUI*

The *GUI* should provide the following mechanisms to the user in order to use the *Simulator*.

- A mechanism to select the mode, namely the debugger mode or default mode in which the simulation can run.
- A mechanism to initialise the clock and set the pace of the clock to one of the available speeds.

- A mechanism to set the timeout for the simulation events.
- A mechanism to start the simulation process.
- A mechanism to graphically view the progress of simulation.
- A graphical representation of *Time Sequence Chart* through which the user can view the status of each object in the system whenever a simulation event is handled.
- A mechanism to invoke the debugger when the simulation is paused.
- A mechanism to invoke the query handler when the simulation is paused.
- A mechanism to invoke the trace analyser when the simulation is paused.
- A mechanism to invoke the reasoning system when the simulation is paused.

The *GUI* should provide the following mechanisms to the user to facilitate query handling.

- A mechanism to view the different types of questions which can be answered by the query handler.
- A mechanism to choose the type of question which can be answered by the query handler.
- A mechanism to view the transition specifications.
- A mechanism to view the transition specifications with its current state as source.
- A mechanism to view the transition specifications with a given state as source.
- A mechanism to view the transition specifications triggered by a given event.
- A mechanism to view the timing constraints.
- A mechanism to view the timing constraints for a given triggering event.
- A mechanism to view the timing constraints for a given constrained event.

The *GUI* should provide the following mechanisms to the user to facilitate trace analysis.

- A mechanism to view the different types of questions which can be answered by the trace analyser.
- A mechanism to choose the type of question which can be answered by the trace analyser.
- A mechanism to view the simulation events which have triggered a transition.
- A mechanism to view the simulation events which have not triggered a transition.
- A mechanism to view the simulation events which have not yet been handled.
- A mechanism to view the simulation events which have occurred during a certain period.
- A mechanism to view the simulation events which have triggered a transition for a given TROM.
- A mechanism to view the simulation events which have triggered a transition for a given TROM during a certain period.
- A mechanism to view the status of the system, subsystem, and TROM simulated at a given point in time.

3.3.3 Requirements of *Reasoning system GUI*

The *GUI* should provide the following mechanisms to the user in order to use the *Reasoning system.*

- A mechanism to view the different types of questions which can be answered by the reasoning system.
- A mechanism to choose the type of question which can be answered by the reasoning system.

- A mechanism to list time intervals during which the system or the TROM object was in a specific state.
- A mechanism to view the times at which the system or the TROM object could get out of a critical state.
- A mechanism to view the set of times at which a particular event was fired.
- A mechanism to view the set of times when a particular event was disabled.
- A mechanism to view the set of times when a particular event was enabled.
- A mechanism to view the set of times when a particular event was scheduled to be fired or disabled later.
- A mechanism to view the assignment vector at a particular time.
- A mechanism to view the reaction vector at a particular time.
- A mechanism to view the set of objects in the system which went into a state during a time interval.
- A mechanism to view the behaviour of the system or of a set of objects in the System.
- A mechanism to view the TROM status during a time interval.
- A mechanism to view the simulation event list of a particular TROM object.
- A mechanism to view all the routes between any two states of a TROM object.
- A mechanism to view a route to a specific state of a TROM.

3.3.4 Requirements of the *GUI* for other components of TROMLAB

This section will give the requirements for the *GUI* to interact with the other components of **TROMLAB** namely *Browser* [Nag99], *Rose-GRC Translator* [Oan99], and the *Verification Assistant* [Pom99].

Interaction with *Browser*

Through *GUI* the Browser can be invoked to perform the following tasks:

1. read and view LSL traits and Larch/C++ specifications from the library;
2. composing new Larch traits with or without reuse of LSL traits from the library;
3. check the syntactic correctness of composed LSL traits as well as analyze some of their properties;
4. run all versions of LSL library traits as well as versions of composed traits against LSL syntax checker;
5. the following basic queries and suitable combinations of them can be posed:
   (a) Retrieve a trait based on name.
   (b) Retrieve a trait based on **includes** (**assumes**) relationship.
   (c) Retrieve a trait based on version number.
   (d) Retrieve a Larch/C++ specification based on name.
   (e) Retrieve a Larch/C++ specification based on **uses** relation.
   (f) Retrieve a Larch/C++ specification based on inheritance information.
   (g) Retrieve traits included in a given TROM specification.
   (h) Retrieve TROMs included in a given subsystem specification.
   (i) Retrieve ports and port links in a given subsystem configuration.
   (j) Retrieve all traits transitively related by **includes** (**assumes**).
   (k) Retrieve all Larch/C++ specifications transitively related through **uses** relation.

Interaction with *Rose-GRC Translator*

The *Rose-GRC Translator* shall run in the *Rose* environment and shall take input from an open *Rose* model. The *Rose* environment can be invoked from the *GUI*, thus promoting the ease of use for the user, to perform the following tasks:

- **Generic reactive class specifications** The user shall be allowed to select the appropriate class diagrams from a list of all the class diagrams in the model, one at a time. A main class diagram shall be selected if all the classes from the subsystem and their port types are shown in that diagram. Separate class diagrams shall be selected if each class and its port types are shown in a separate class diagram.

- **Subsystem Configuration Specification** Given that the model may contain several subsystems, the user shall be allowed to select the appropriate collaboration diagrams, from a list of all collaboration diagrams in the model. The user shall be allowed to enter a name for each subsystem corresponding to the selected diagram. The user shall be allowed to skip translation of a subsystem configuration.

- **Message Sequencing** The user shall be allowed to select the appropriate sequence diagram from a list of all sequence diagrams in the model, one at a time. The user shall be allowed to skip translation of a sequence diagram. The user shall be allowed to enter a time of occurrence for each message in the sequence diagram.

- **Reactive class specifications** *GUI* will receive from the *Rose-GRC Translator* an output a text file containing class specifications for all the generic reactive classes (GRC) that are found in the selected class diagram.

- **Subsystem Configuration Specification** *GUI* will receive from the *Rose-GRC Translator* an output a text file containing the configuration specification of the subsystem modeled in the selected collaboration diagram.

- **Message Sequencing** *GUI* will receive from the *Rose-GRC Translator* an output text file containing an ordered list of messages together with the sending and receiving objects and the time of occurrence, from a selected sequence diagram.

Interaction with *Verification Assistant*

Verification Assistant [Pom99] is a tool within TROMLAB environment for an automated axiom generation based on the methodology described in [MA99]. This tool

can be used to generate various axioms, after creating the model using the *Interpreter*. The following are the requirements of the *GUI* for *Verification Assistant*:

- *GUI* should provide facility for the user to generate the transition and time constraint axioms.
- *GUI* should provide facilty for the user to generate the synchrony axioms tha correspond to the subsystem of the model.
- *GUI* should provide a mechanism to view the generated axioms.

The following chapters discuss in detail *GUI* design for *Interpreter*, *Simulator*, and the *Reasoning System*. *GUI* design for interaction with the other components is rather simple and thus is not discussed.

Chapter 4

Design and Implementation of the Modified *Interpreter* and *Simulator*

In this chapter we compare the new designs of the *Interpreter*, and the *Simulator* with this old designs to emphasize the significant improvements that were made according to the requirements given in the previous chapter. We also discuss the tools which were used to implement the *Interpreter*.

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 relationships 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.

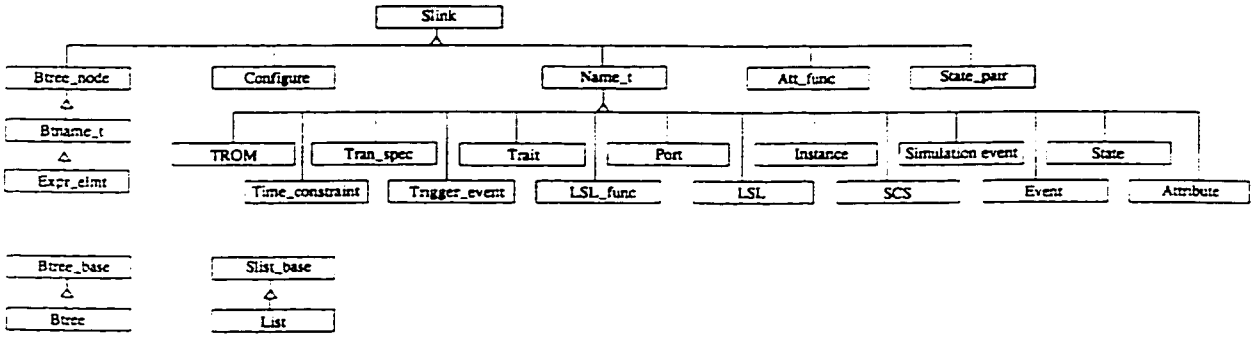


Figure 11: *Interpreter* Class diagram (Old)

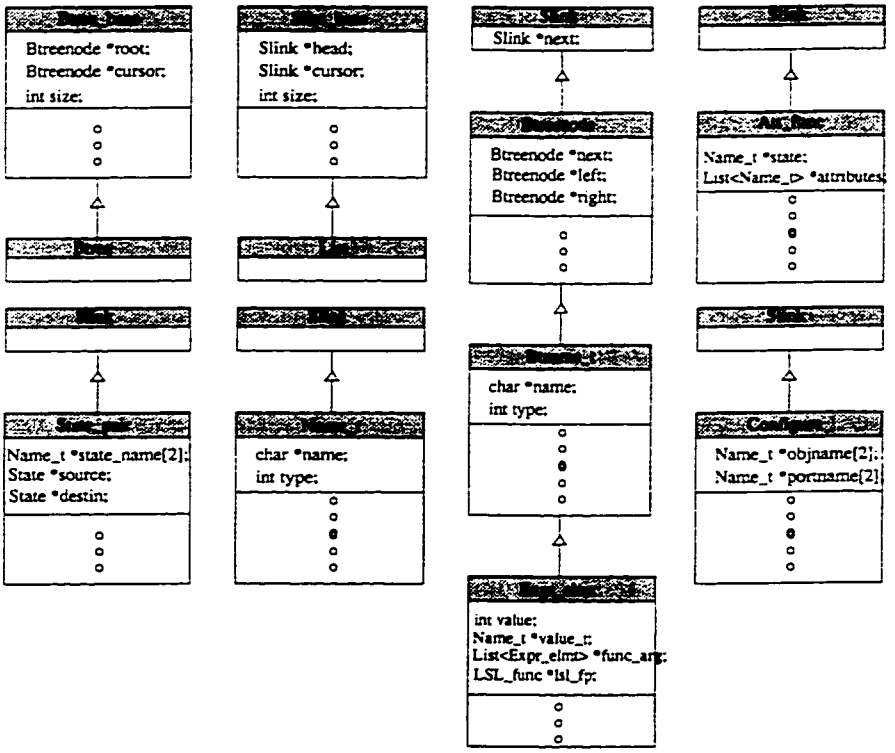


Figure 12: *Interpreter* Class diagram - Detailed (Old)

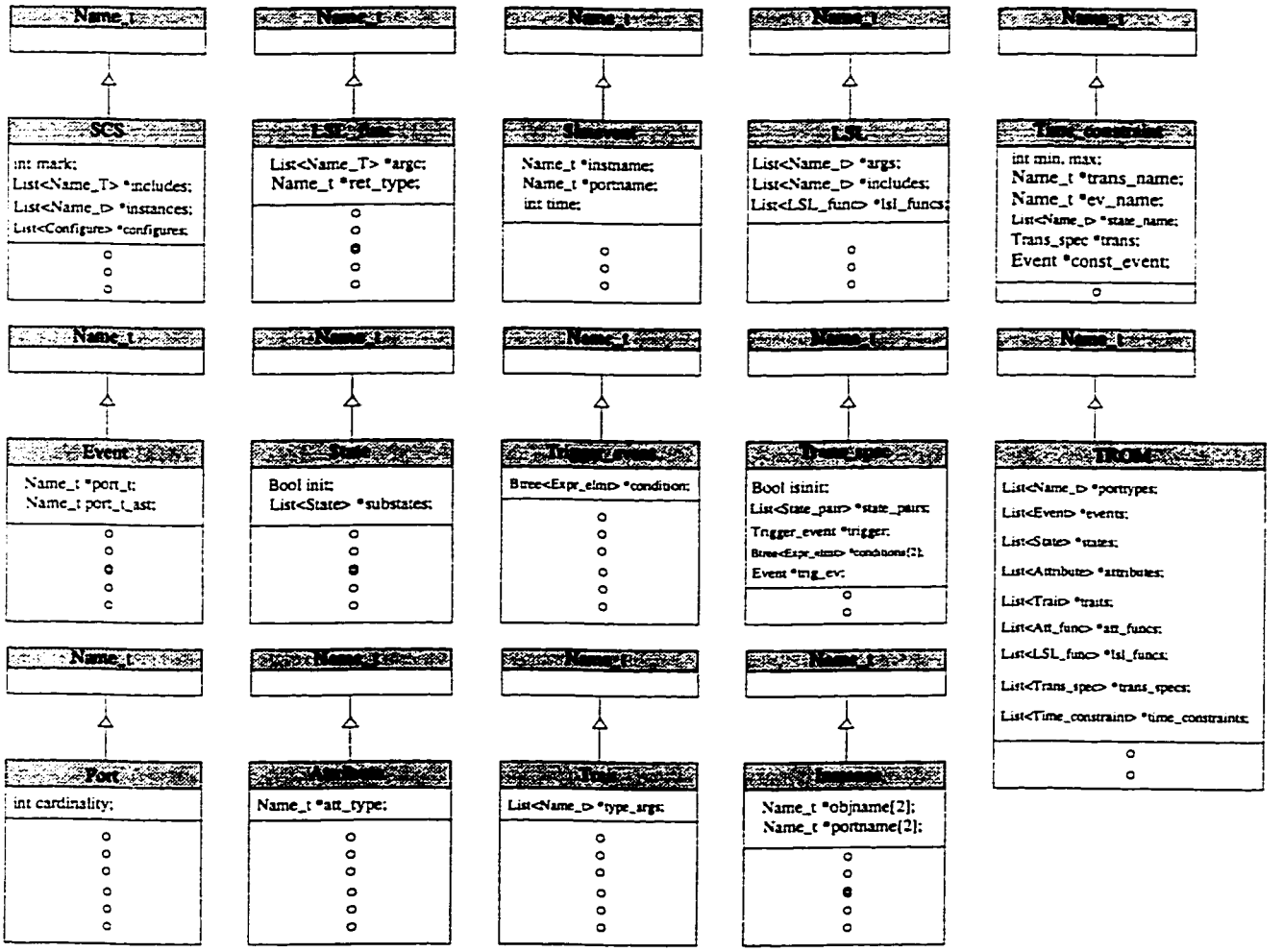


Figure 13: *Interpreter* Class diagram - Detailed (Old)

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 tiers, 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.

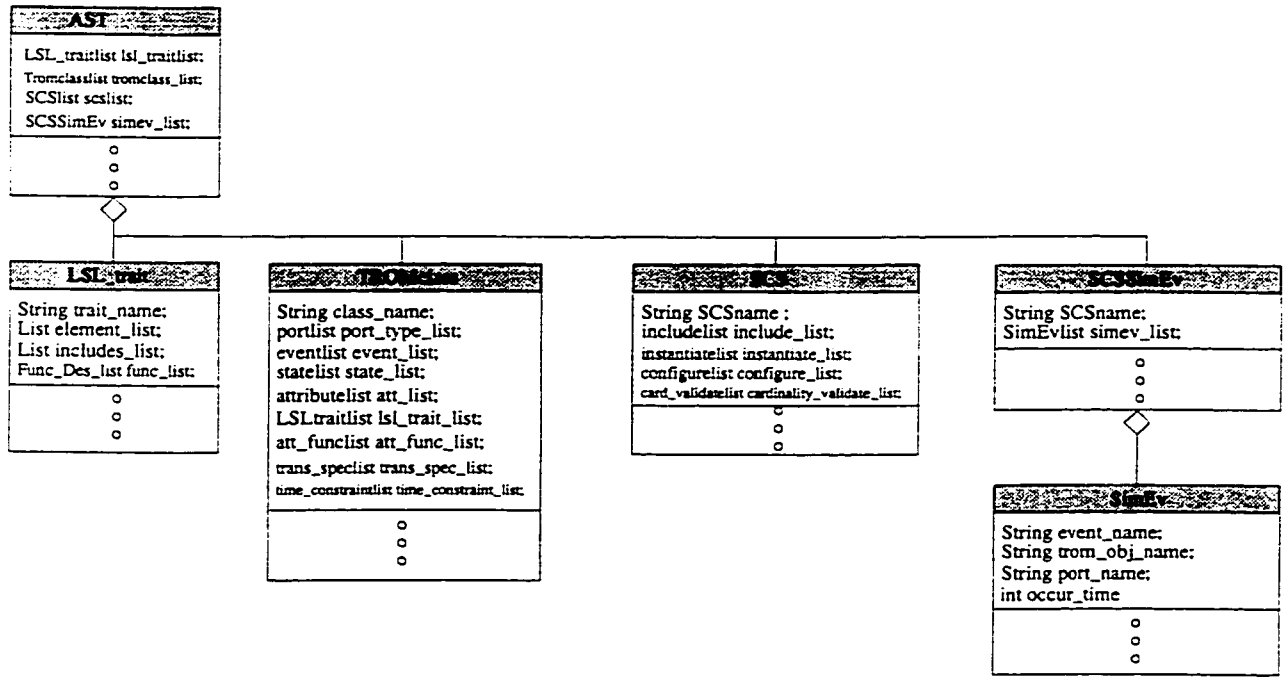


Figure 14: *Interpreter* Class diagram (New)

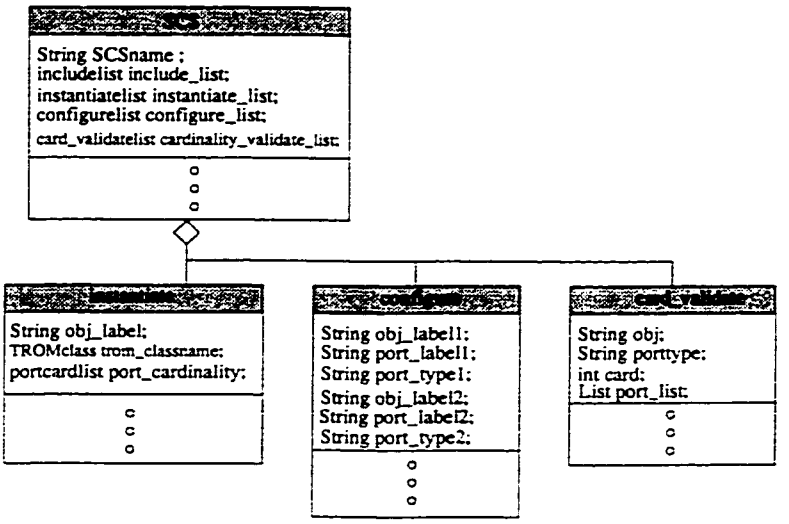


Figure 15: *Interpreter* Class diagram - SCS (New)

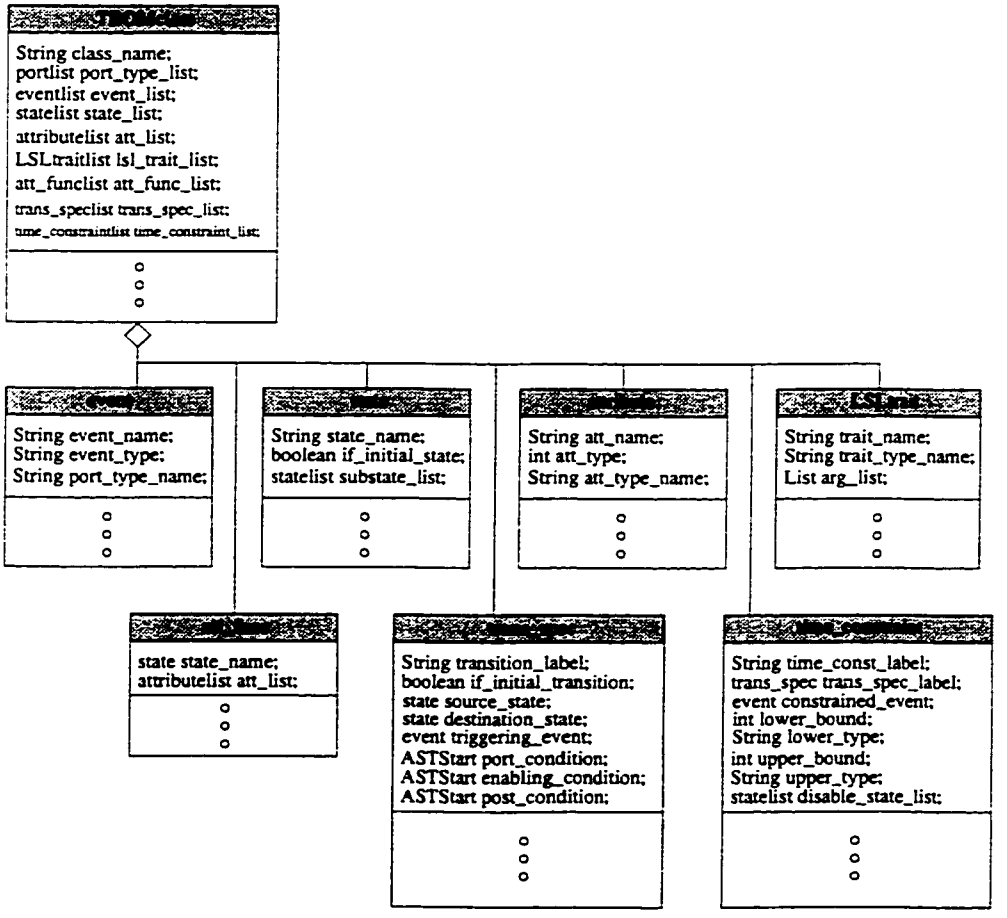


Figure 16: *Interpreter* Class diagram - TROMclass (New)

4.1.2 *Simulator*

Class diagram (old Version): 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 the Figure 17, Figure 19, and Figure 18.

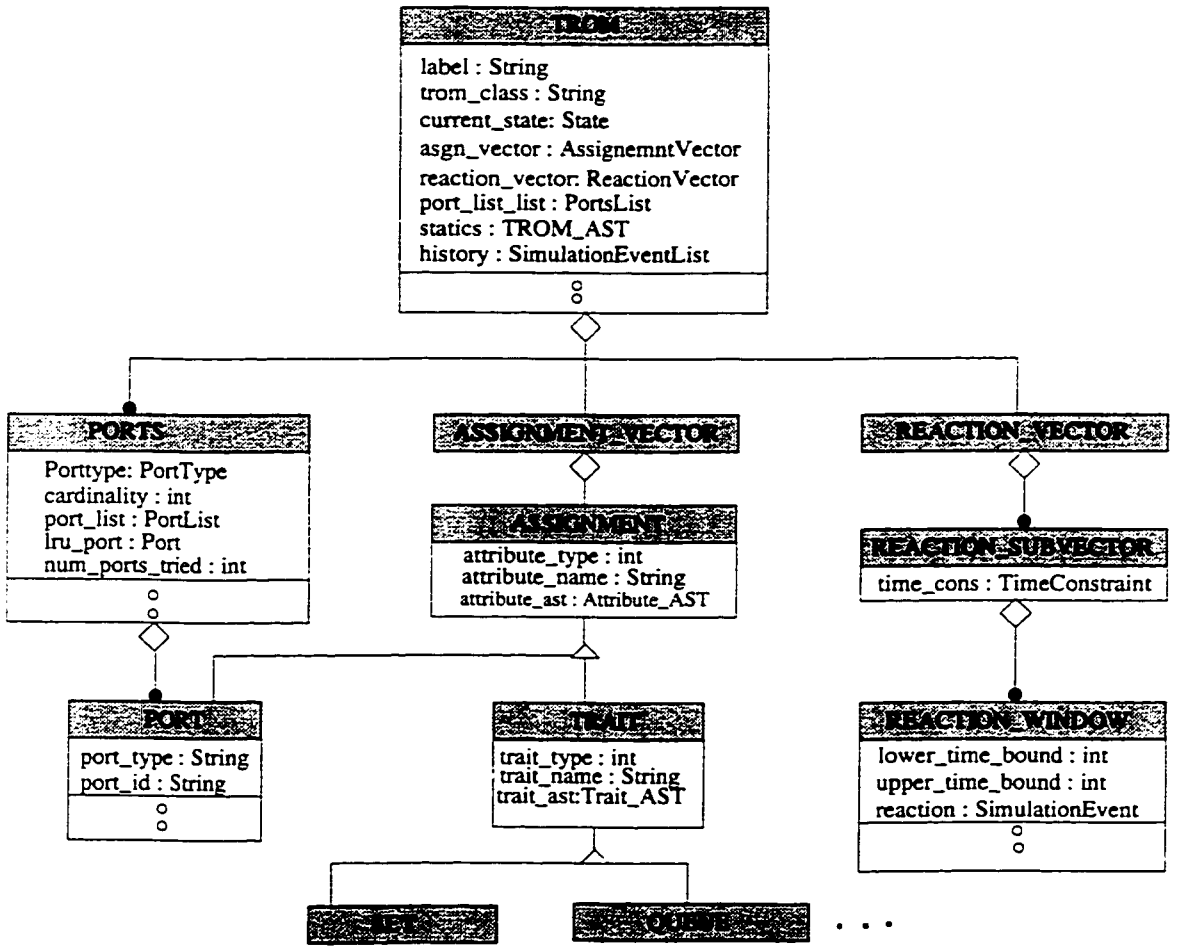


Figure 17: *Simulator* Class diagram - TROM class diagram(New)

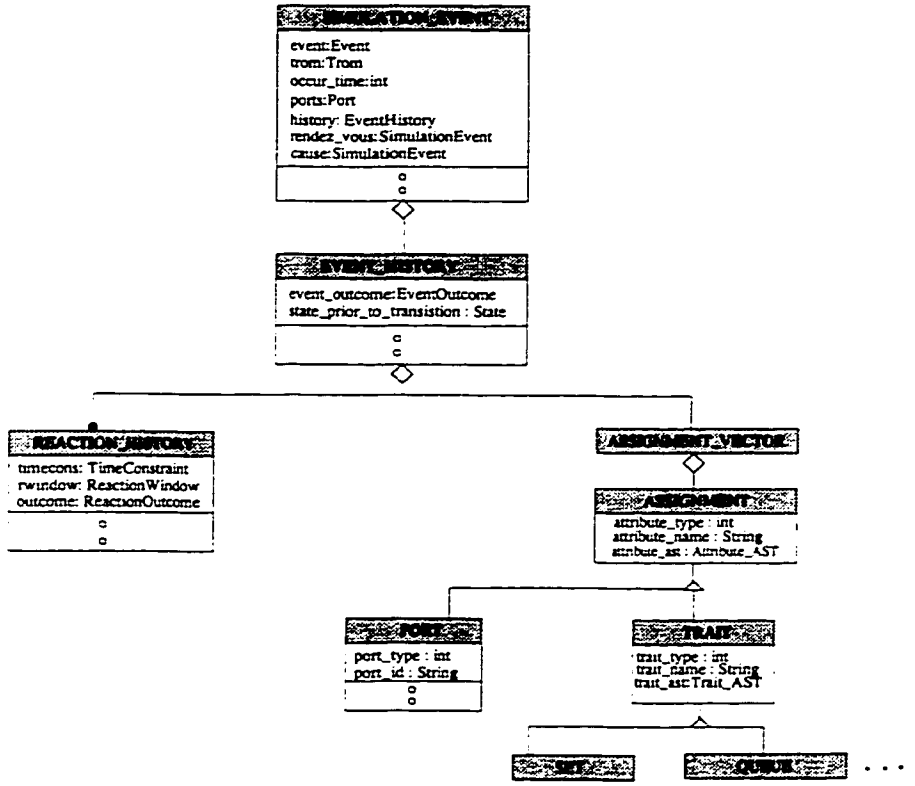


Figure 18: *Simulator* Class diagram - Simulation Event Object model (New)

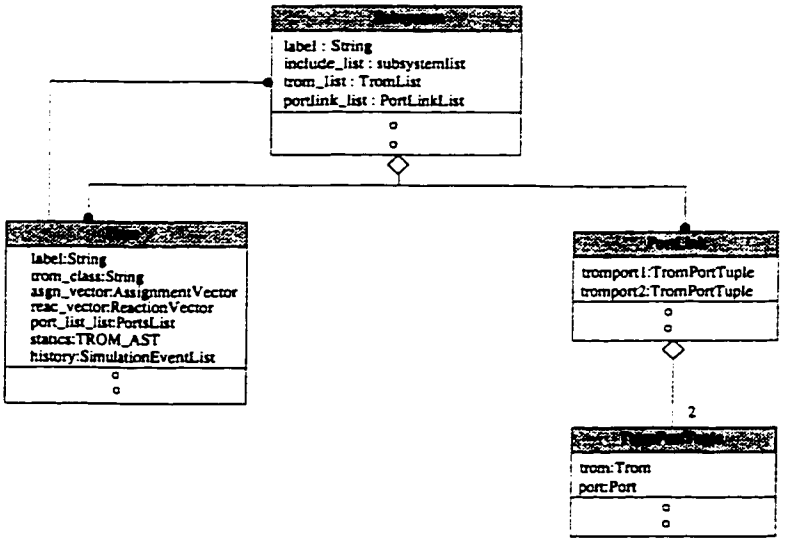


Figure 19: *Simulator* Class diagram - Subsystem Object model (New)

4.2 Language of choice

We have chosen Java as the language of implementation for 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 recognize 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 itself.

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 analyze its behavior.

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 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 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 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:

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:

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

Figure 20: Simulation event list

- 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 analyzer.

7. Trace analyzer: Trace analyzer 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

Graphical User Interface : Design and Implementation

A *graphical user interface GUI* has been recognised as an important requirement for the usability of the TROMLAB system. The overriding objective was to provide a user-centered design of *GUI* of TROMLAB, which makes the system easy to learn and use, and provides safe and effective ways for the user to interact with TROMLAB in a task-oriented manner. A window-based interactive user interface is built to support user interaction. We first present the significant aspects of *GUI* design issues, and then focus on the detailed design of different *GUI* components. This design has been fully implemented using Java programming language. All design diagrams presented in this chapter are drawn according to the UML standard notation.

5.1 *GUI* design issues

The fundamental principle in user interface design is that the interface must be designed to suit the needs and abilities of the individual users of the system. The users of our TROMLAB system are software designers with some familiarity with formal specification languages, OOD, and TROM semantics. The important features in the design of user interface are as follows:

- Users can use multiple sources on screen at once to carry out a task.
- Users may be able to interact with any one of several multiple views of one item of interest on screen at the same time, for example can use *Rose-GRC Translator*

to create visual models and generate respectively the formal specifications and check them for syntactical and semantic correctness using *Interpreter GUI.*

- Users are allowed to specify objects in the screen by pointing, selecting, dragging, and dropping them.

Our design and implementation efforts were focussed on the following aspects:

- **Learnability and Throughput** The menu buttons, directory tree structure, and dialog boxes help users to reach a specified level of use performance within a short time. The drag and drop feature provided in the directory tree structure facilitates the speed of task execution, at the same time making it easy to use.
- **Consistency** The *GUI* design of windows are consistent, for example the window titles, menu bars, scroll bars, and colours are used in a consistent way.
- **Familiarity** Concepts and terminology that the user is already familiar with a re-incorporated into the interface, such as in menu button names, labels, and window messages.
- **Control** All the messages are positive, polite, and concise to make user feel control over the system.
- **Scalability** The *GUI* was designed and implemented to work well for a small process or collection of data, as well as large sized collections of data.
- **Modifiability** The *GUI* design supports future expansions of other TROMLAB components without requiring major design changes to it.

5.2 Detailed Design of *GUI*

In this section we present the detailed design of various *GUI* components namely *Interpreter GUI*, and *Simulator GUI. Reasoning system GUI* can run in conjunction with *Simulator*, forming part of the *Simulator GUI.* We provide use case diagrams, and object diagrams for the different *GUI* components.

5.2.1 *Interpreter GUI*

A use case diagram was created as part of the analysis of the user requirements for the *Interpreter GUI*, and is illustrated in Figure 21.

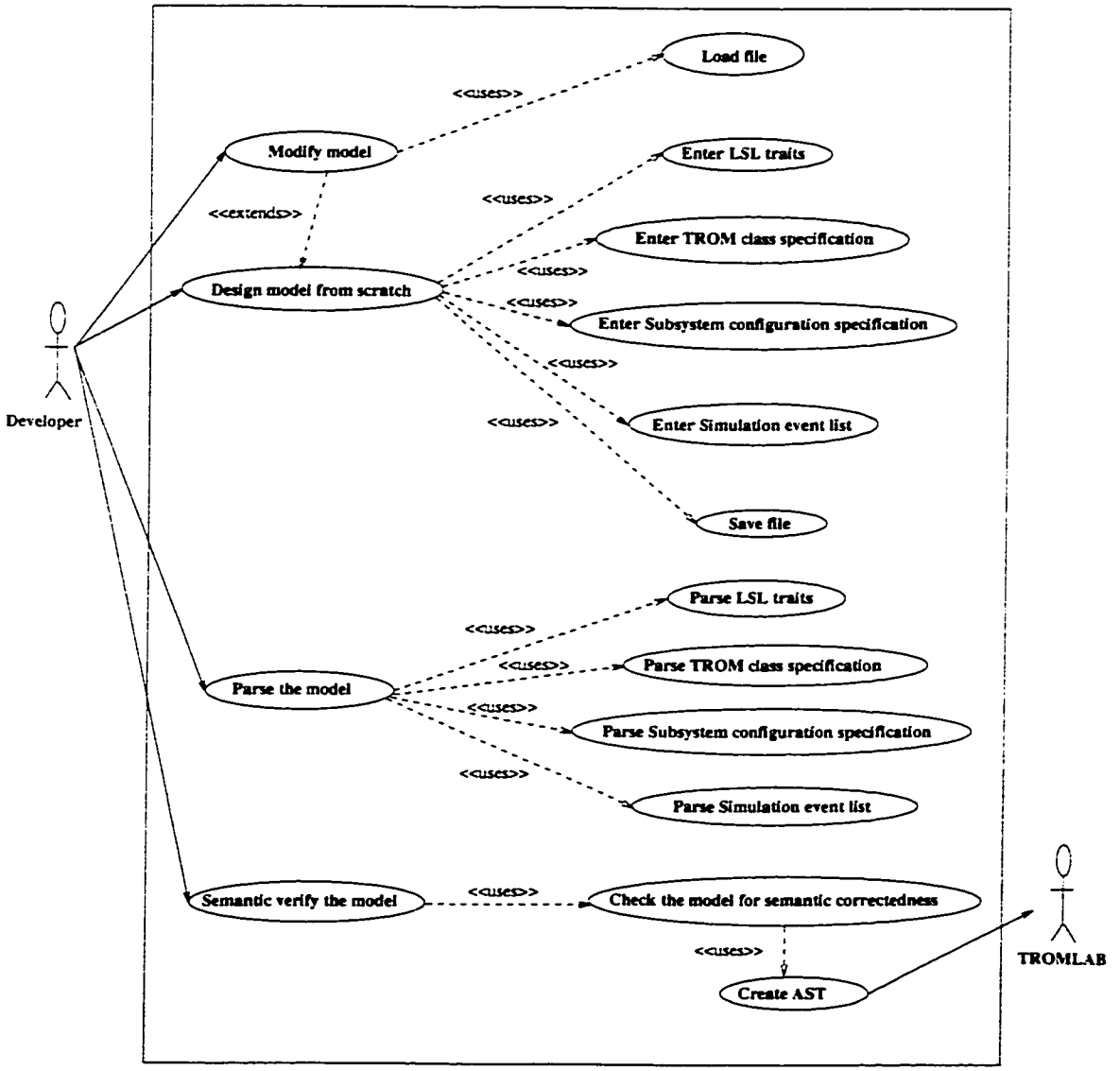


Figure 21: Use Case diagram for *Interpreter GUI*

This use case diagram contains two actors: Developer, and TROMLAB. The diagram shows the main use cases(Design model from scratch, Modify model, Parse model, and Semantic verify the model). The Developer designs the model, and finally the internal structure of the model i.e. AST is created which is sent to TROMLAB system.

During the Object modelling phase we have designed object classes, object representations, designed associations, organised object classes using aggregation and inheritance, and grouped classes into modules based on the closed coupling and related functions.

In particular, the key classes where Mainwindow, InterpreterPane, DirectoryTreePane, DraggableTree, DroppableTextArea, and MyTreeModel. Figure 22 illustrates the object diagram for the *Interpreter GUI*.

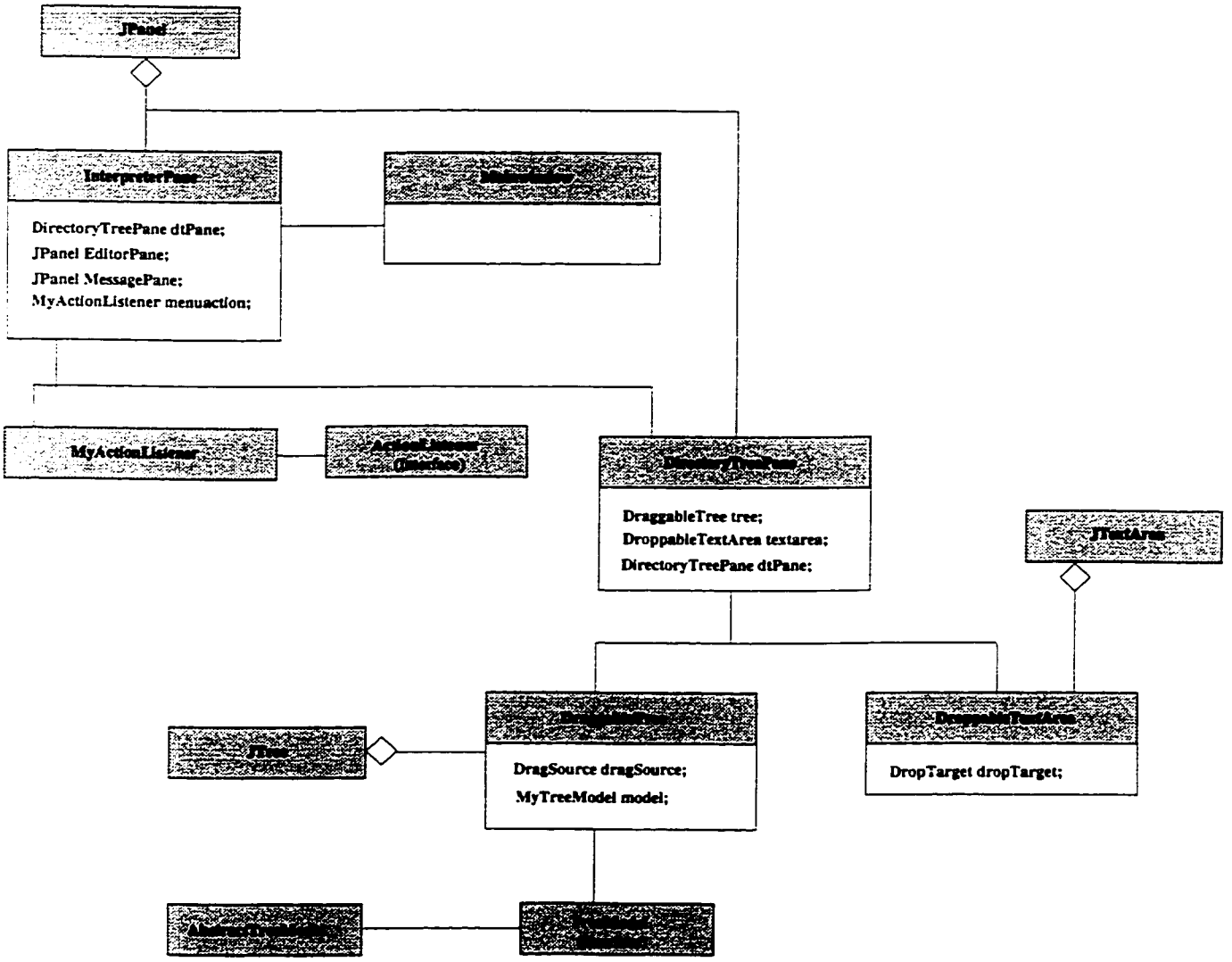


Figure 22: Object diagram for *Interpreter GUI*

In Appendix C, we give a set of VDM specification for describing the tasks of *Interpreter GUI* interface.

5.2.2 *Simulator GUI*

A use case diagram was created as part of the analysis of the user requirements for the *Simulator GUI*, and is illustrated in Figure 23.

This use case diagram contains six actors: Developer, Simulator, Debugger, Query Handler, Trace Analyser, and Reasoning system. The diagram shows the main use cases(Start simulation, Start debugging, Start query handler, Start trace analyser, and Start reasoning system).

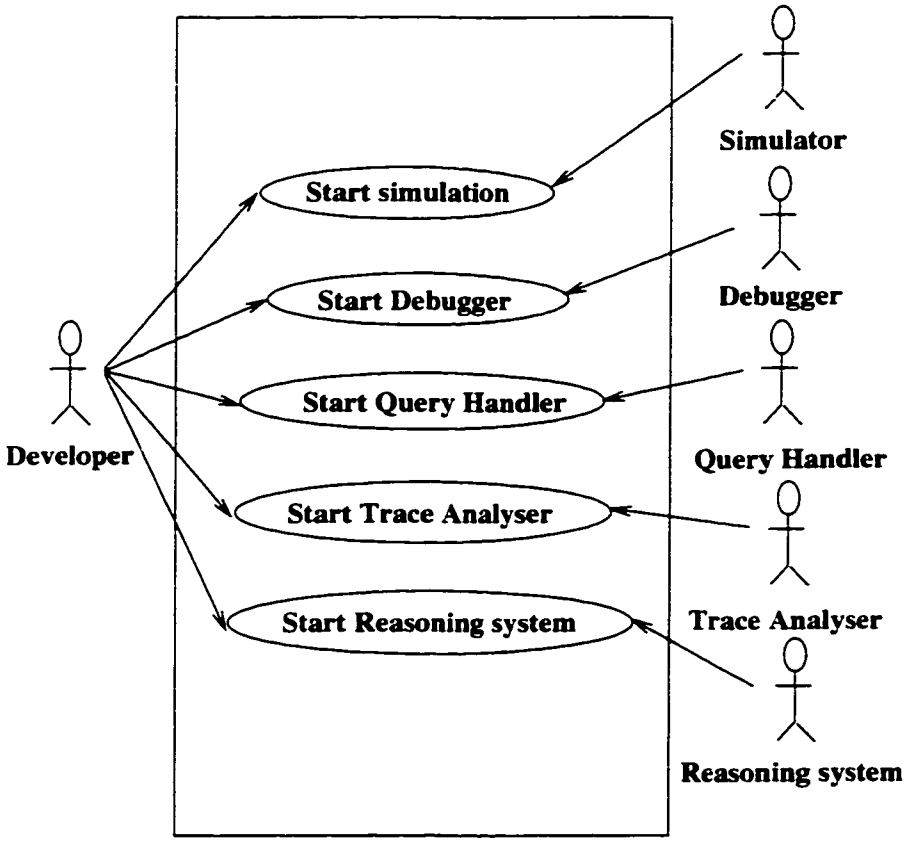


Figure 23: Use Case diagram for *Simulator GUI*

During the Object modelling phase we have designed object classes, object representations, designed associations, organised object classes using aggregation and inheritance, and grouped classes into modules based on the closed coupling and related functions.

In particular the key classes where Mainwindow, SimulatorPane, DebuggerPane, QueryHandlerPane, TraceAnalyserPane, ReasoningSystemPane, and MyTableModel. Figure 24 illustrates the object diagram for the *Simulator GUI*. In Appendix C, we

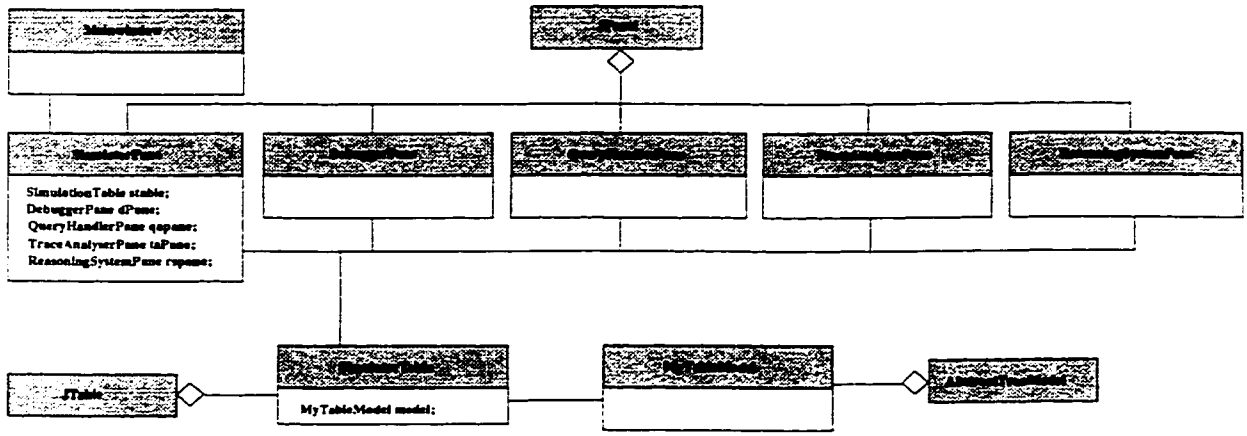


Figure 24: Object diagram for *Simulator GUI*

give a set of VDM specification for describing the tasks of *Simulator GUI* interface.

5.3 *GUI* Implementation

The *GUI* for TROMLAB was implemented using Java programming language(JDK 1.2.1), with the work done under Sun workstation running Solaris 7. JDK 1.2.1 kit provides Swing library which provides a complete set of graphical widgets designed to implement user interfaces.

The most remarkable feature of Java Swing components is that they are written in 100 percent Java and do not depend on peer components, as most AWT components do. This means that a Swing button or text area will look and function identically on Macintosh, Solaris, Linux, and Windows platforms. This design eliminates the need to test and debug applications on each target platform.

All the operations specified in the user requirements are implemented with all pre- and post conditions checked. The *GUI* was implemented according to the design efforts described in the previous section. The windows snapshots of the Interpreter GUI, and *Simulator GUI* are given in Figure 25 and Figure 26, followed by an explanation.

5.3.1 Snapshot of *Interpreter GUI*

There are three panels in the window(Figure 25): The panel on the left has a directory tree structure which displays the valid formal specification files available in the disk; The panel on the right is the text editor which allows the user to design the formal specifications of the system; The bottom panel displays the messages when user checks for syntax and semantics of the formal specification file.

User can drag the existing formal specification file from the directory tree panel and drop it on the text editor panel. As soon as the user drops the dragged formal specification file on the text editor panel, it is displayed in the editor. User is also provided with another option to open the formal specification file by choosing the menu item under the File menu, or by clicking the right mouse button anywhere in the window. User is allowed to create a new formal specification by using the editor provided, and can save the composed formal specification file.

The *Parse button* can be used to activate the syntactic analysis of the formal specification file. When the syntax analysis is completed, the message is displayed in the message panel. The error messages are meaningful, and guide the user to correct the error(s) in the formal specification file. Once the user has syntactically checked all the four formal specification files, namely TROM class specification(s), Subsystem Configuration Specification, LSL trait, and Initial simulation event list, semantic verification can be done by clicking the semantic verify menu item. The semantic verify menu item becomes active only when the user has successfully parsed all the above mentioned formal specification files. This menu item is part of the *Semantic Analysis* menu.

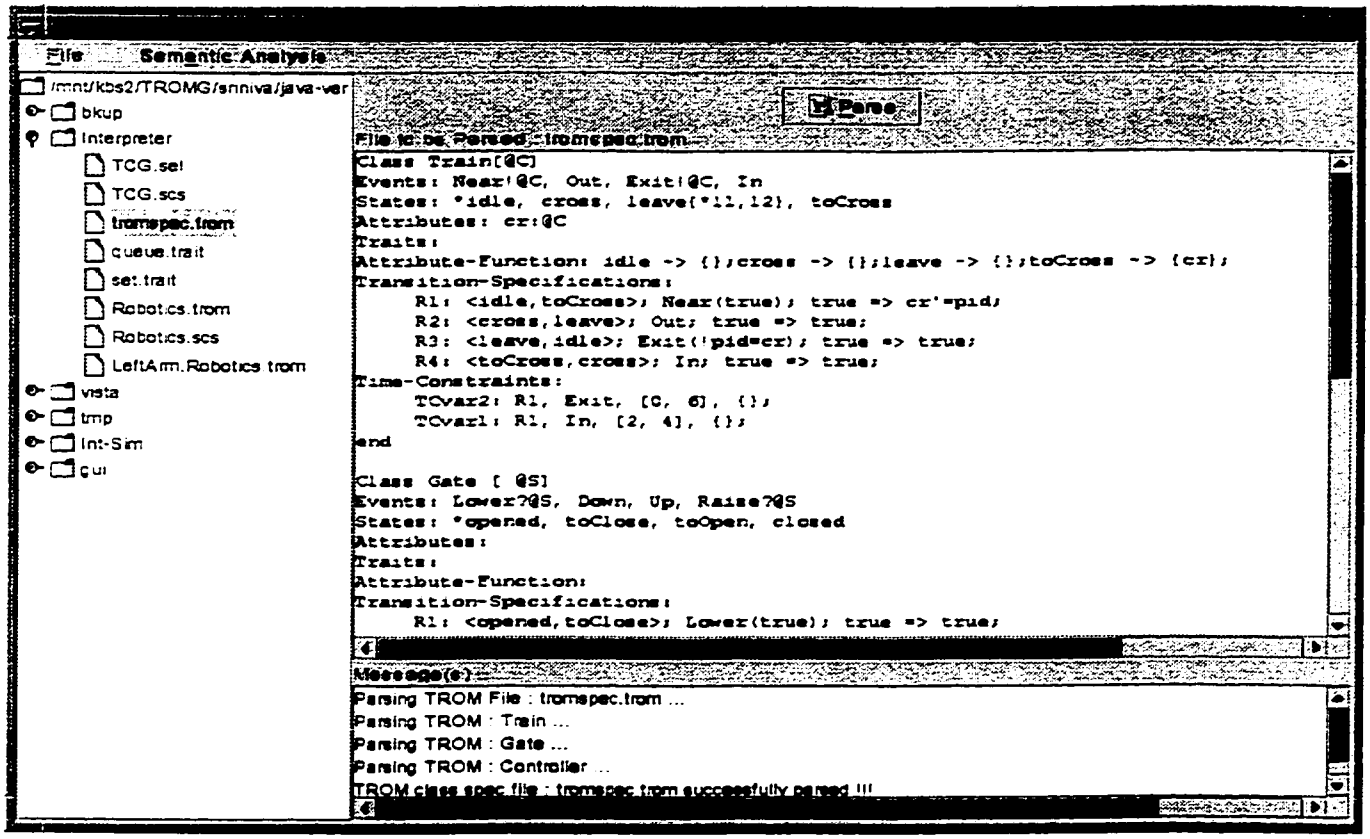


Figure 25: Window of *Interpreter GUI*

5.3.2 Snapshot of *Simulator GUI*

Figure 26 shows the snapshot of the *Simulator's GUI*. Figure 27, Figure 28, Figure 29, and Figure 30 show the snapshots of *Debugger GUI*, *Query Handler GUI*, *Trace Analyser GUI*, and *Reasoning System GUI* respectively. Before the user can start the simulation of the designed system, simulation parameters which include *Debug option*, *Pace option*, and *Timeout value* should be provided. In order to do that, the user can make use of the radio buttons to set the debug option and pace option, and the textbox is used to provide the simulation time out value. Only after providing these values the *Start button* will be enabled to start the simulation.

The simulation can be viewed graphically by the user in a tabular form which is in the *Simulator GUI*. The columns in the table represent the objects instantiated in the system. Before the simulation is started the columns are constructed from the object names provided in the subsystem configuration specification. The rows in the table represent simulation events handled at that particular simulation time. So, a table cell represents the state of the object in the system for a handled simulation event. The table entries projects the change in the states of the objects in the system as time progresses. As soon as the simulation starts the table entries start to appear and a row is added to it whenever a new simulation event is handled.

If the simulation is started in a debug mode, the user will be allowed to invoke *Debugger GUI*, *Query Handler GUI*, *Trace Analyser GUI*, and *Reasoning System GUI* by clicking the respective menu items. In debug mode the user is allowed to query the system, find the state of the system to analyse the cause for its current state, view history of the simulation, and inject a simulation event etc. This allows the user to debug the design and verify whether the system is behaving according to the design.

Debugger provides a set of queries which allow the user to get information about the current status of the simulation. For example, the query *Display system status* displays the current status of the simulated system. It displays the current state of every TROM object, the assignment vector, and the reaction vector of the system. *Query handler* provides a set of queries which allow the user to get static information of the system. For example, the query *Display transitions for given Trom* displays all

the transitions of the TROM object given by the user. *Trace analyser* provides a set of queries which allow the user to get information about the history of the simulation. For example, the query *Display simulation events causing no transition* displays the simulation events which were *disabled. Reasoning system* provides a set of queries which allow the user to get the following types of information:

- History queries: The query *why does the system goes from one state to another state* is an example of a history query. The response to the query allows the user to understand the reasons which caused the transitions that lead the system to go from one state to another state.

- Hypothetical queries: The query *what if we insert an event* is an example of a hypothetical query. The response to the query allows the user to analyse the consequences of inserting a new simulation event.

- Reachability queries: The query *show all the routes between any two states of a TROM* is an example of a reachability query. The response to this query is a display of all possible acyclic routes between any two given states of a TROM object.

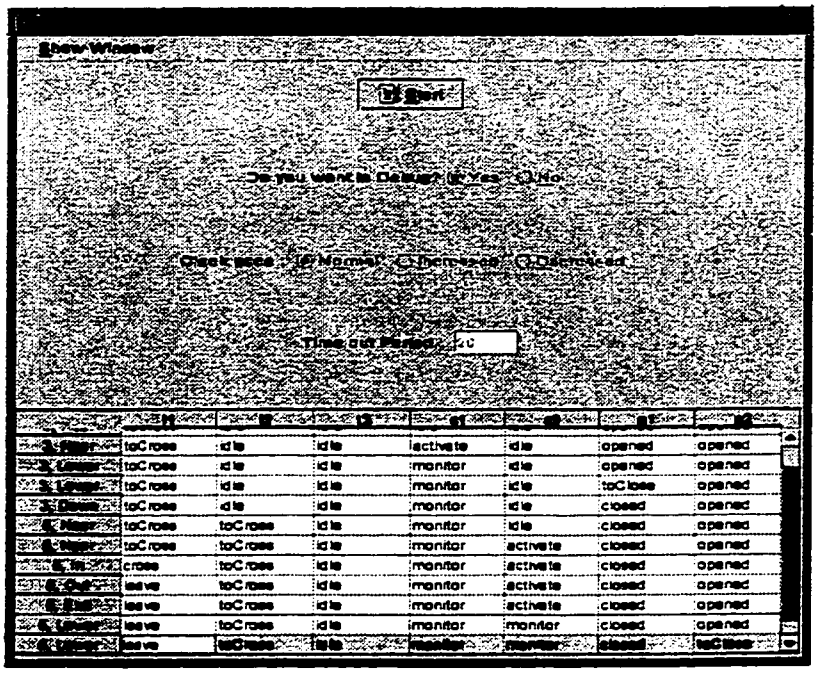


Figure 26: Window of *Simulator GUI*

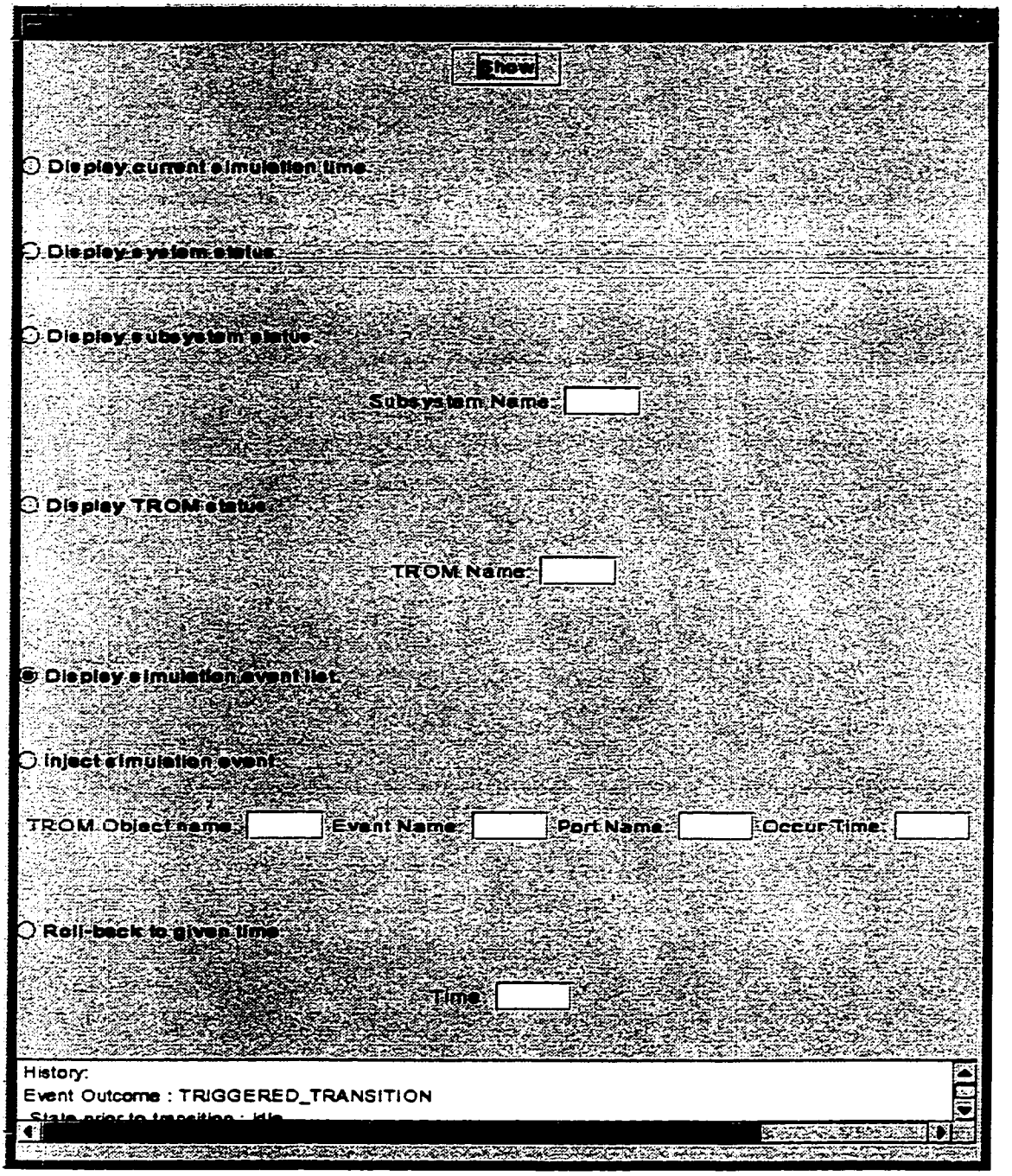


Figure 27: Window of *Simulator - Debugger GUI*

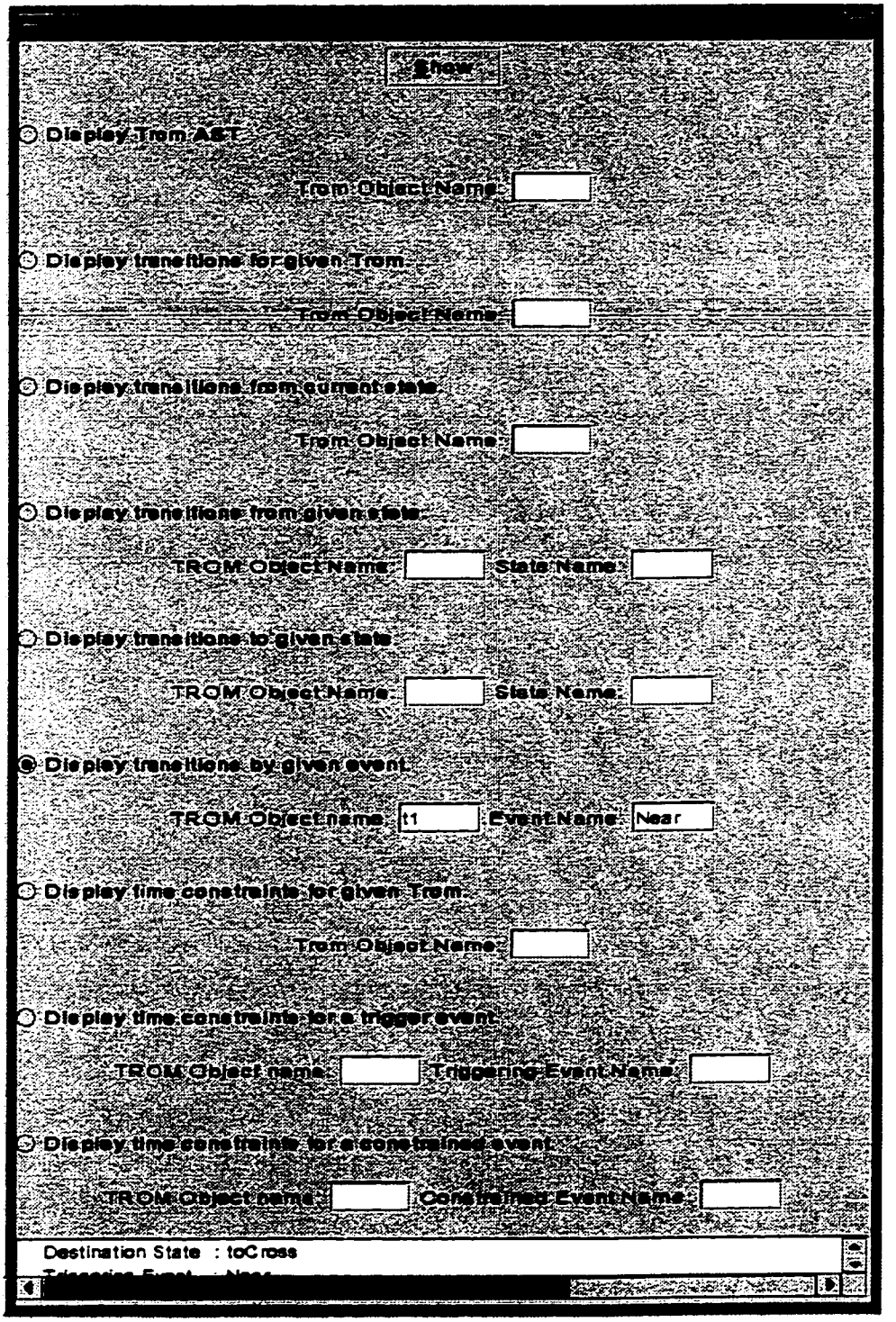


Figure 28: Window of *Simulator - Query Handler GUI*

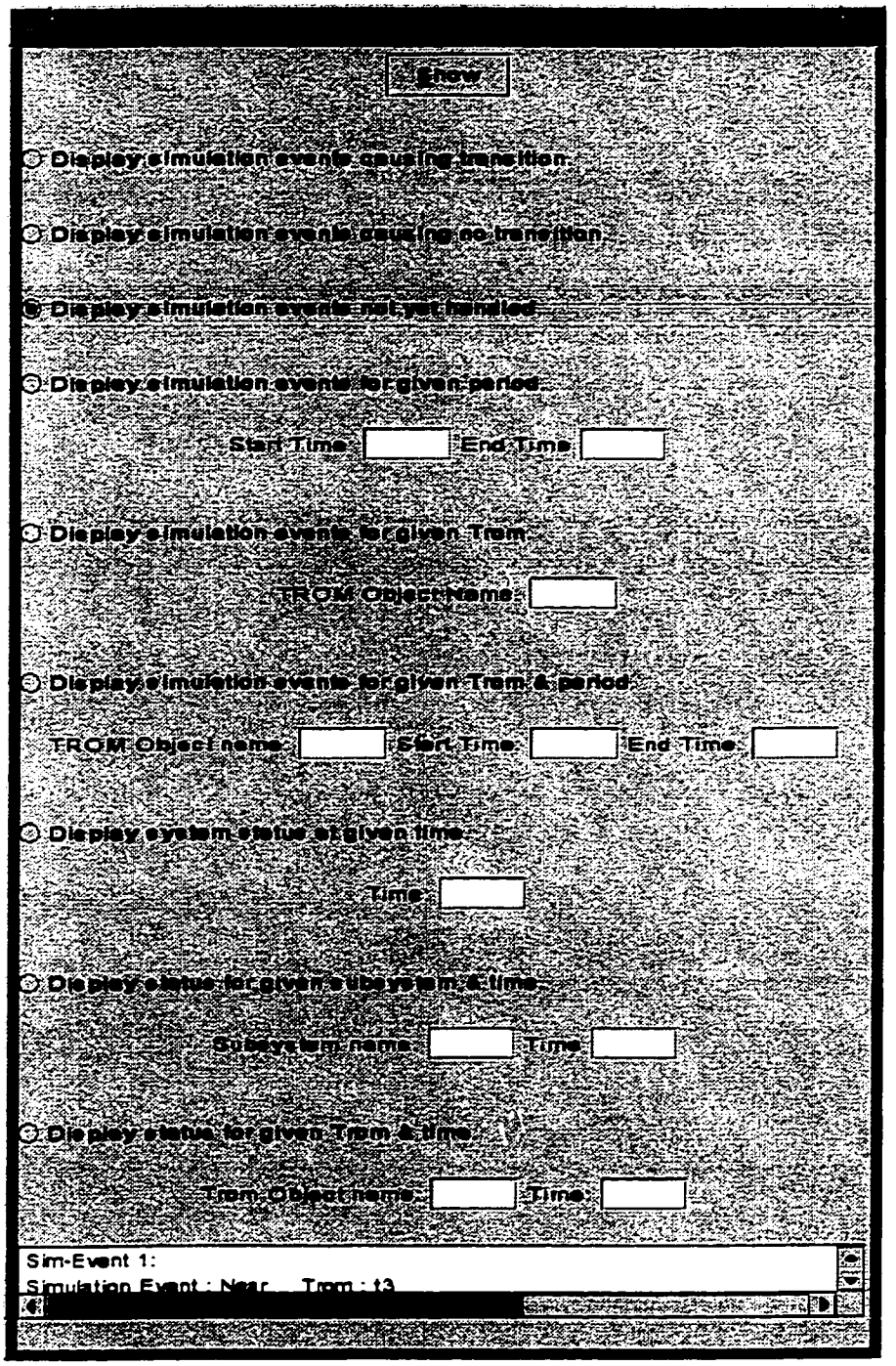


Figure 29: Window of *Simulator - Trace Analyser GUI*

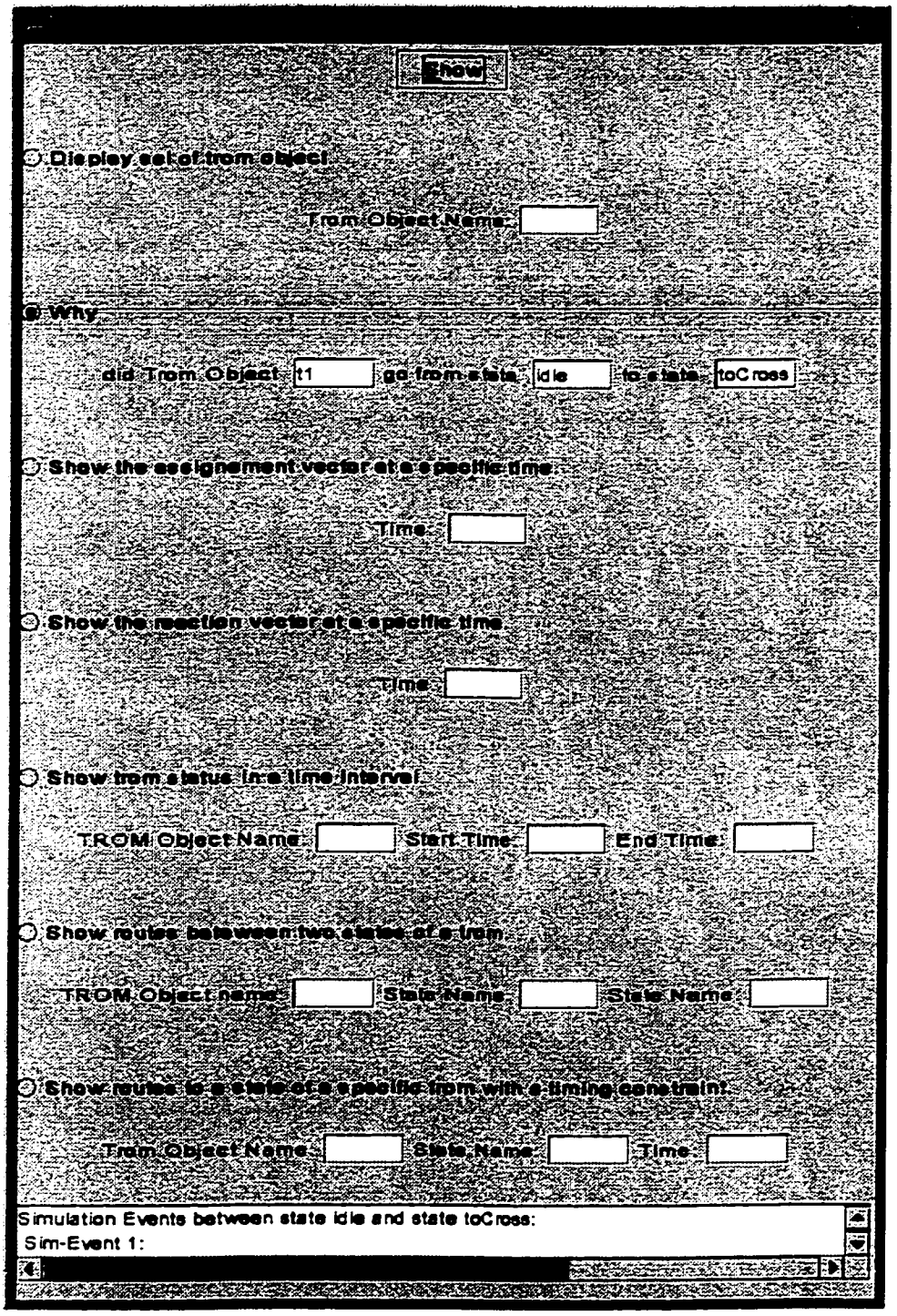


Figure 30: Window of *Simulator - Reasoning System GUI*

Chapter 6

Case Study : Robotics Assembly example

6.1 Introduction

This chapter demonstrates the GUI application for a *Robotics Assembly* problem. The problem and its model will be described informally and formally and then the tool's application will be shown.

6.2 Problem Description

6.2.1 Informal Problem Description

We abstract *robots* from mechanical objects to functional units. The assembly environment consists of a *robot* with two arms, a *conveyor 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. We model each component as a GRC with port types and attributes. The *User* has one port type $@VS$ to communicate with the *Vision System* when parts are placed on the *belt*. The *belt* has two port types: port type $@V$ to receive a message from the *Vision System* when a *part* has been sensed; and port type $@R$ to receive messages from the *Robot* when a *part* has been picked. The *Vision System* 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 *Vision System* when a *part* has been recognised; and port type $@D$ to inform the *belt* that a *part* has been picked. Figure 31 shows the TROM classes, with respective port types in the *Robotics Assembly* system.

The dynamic behaviour of the reactive objects are captured in the statechart diagrams shown in Figure 33, Figure 39, Figure 36, and Figure 42. The assembly system, consisting of two users, one *vision system*, one *belt*, and one *robot*, is described in the collaboration diagram in Figure 45. The formal specifications are shown in Figures 32, 38, 35, and 41. The LSL trait *PartType[Part]* is an abstract enumerated type for defining *cup* and *dish* parts. Table 6.2.1 describes the situations captured by the states for the robot manipulator in Figure 42.

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.

6.2.2 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*.

Figure 31 shows the TROM classes, with respective port types in the Robotics assembly system.

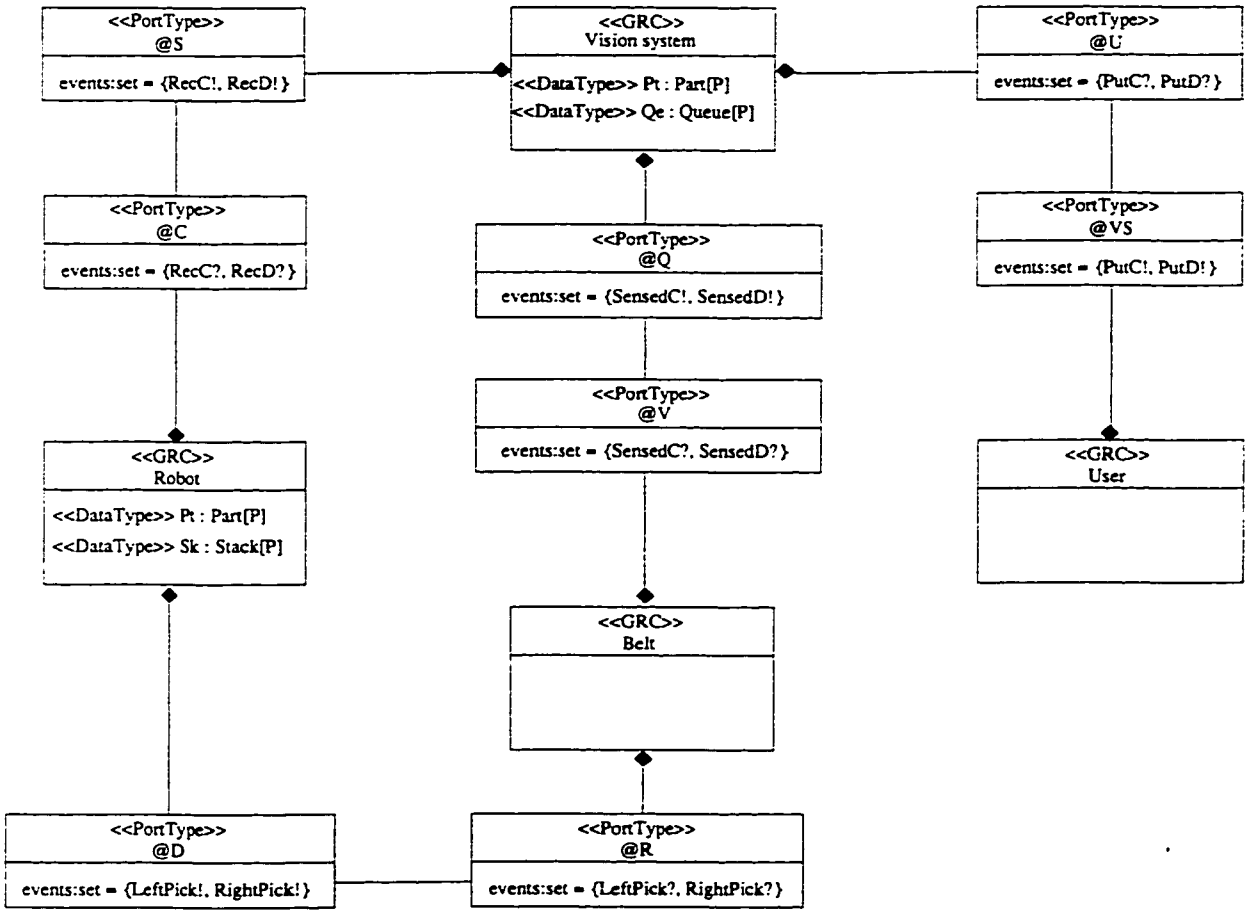


Figure 31: Robotics System Class diagram

6.2.3 Formal Problem Description

In this section we 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. Figure 32, Figure 33, and Figure 34 show the textual representation, state machine representation, and the UML model of the *User* class.

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 32: User TROM class - Textual representation

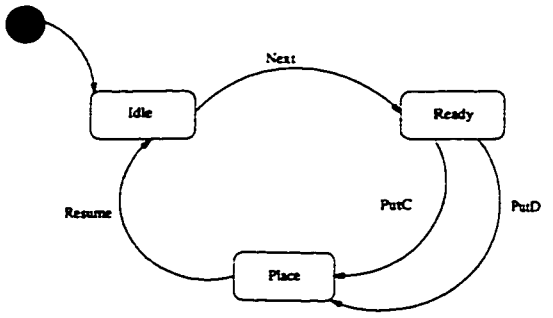


Figure 33: User TROM class - State machine representation

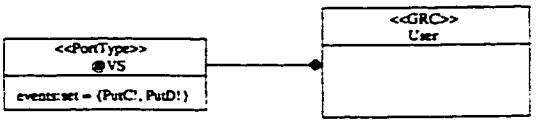


Figure 34: User TROM class - UML model

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 *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. Figure 35, Figure 36, and Figure 37 show the textual representation, state machine representation, and the UML model of the *Vision System* class.

```
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 35: Vision system TROM class - Textual representation

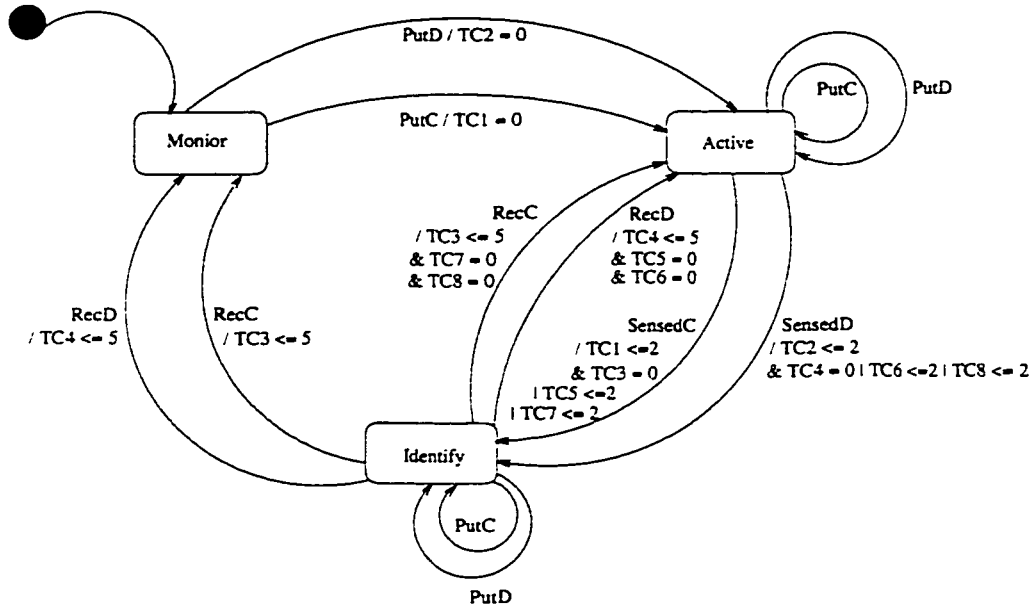


Figure 36: Vision system TROM class - State machine representation

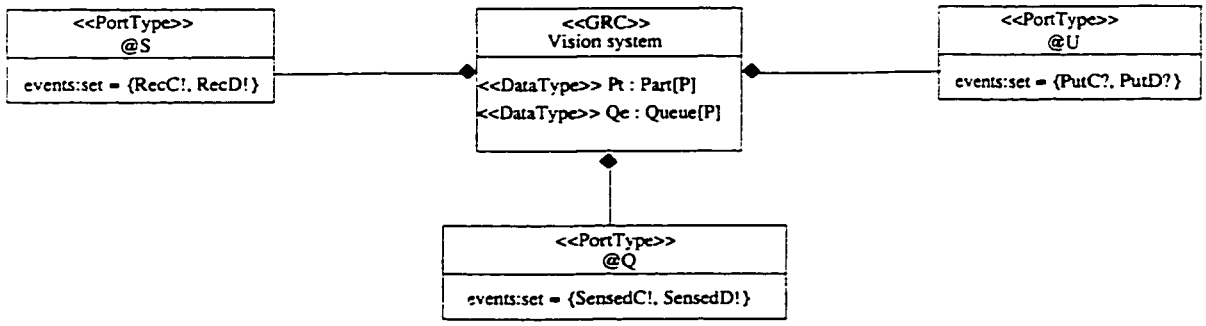


Figure 37: Vision system TROM class - UML model

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. Figure 38, Figure 39, and Figure 40 show the textual representation, state machine representation, and the UML model of the *Belt* class.

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 38: Belt TROM class - Textual representation

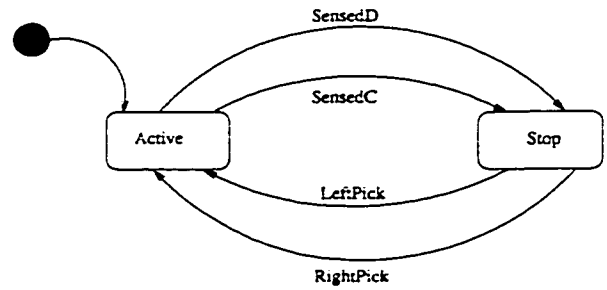


Figure 39: Belt TROM class - State machine representation

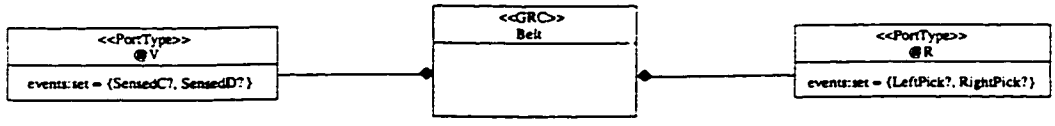


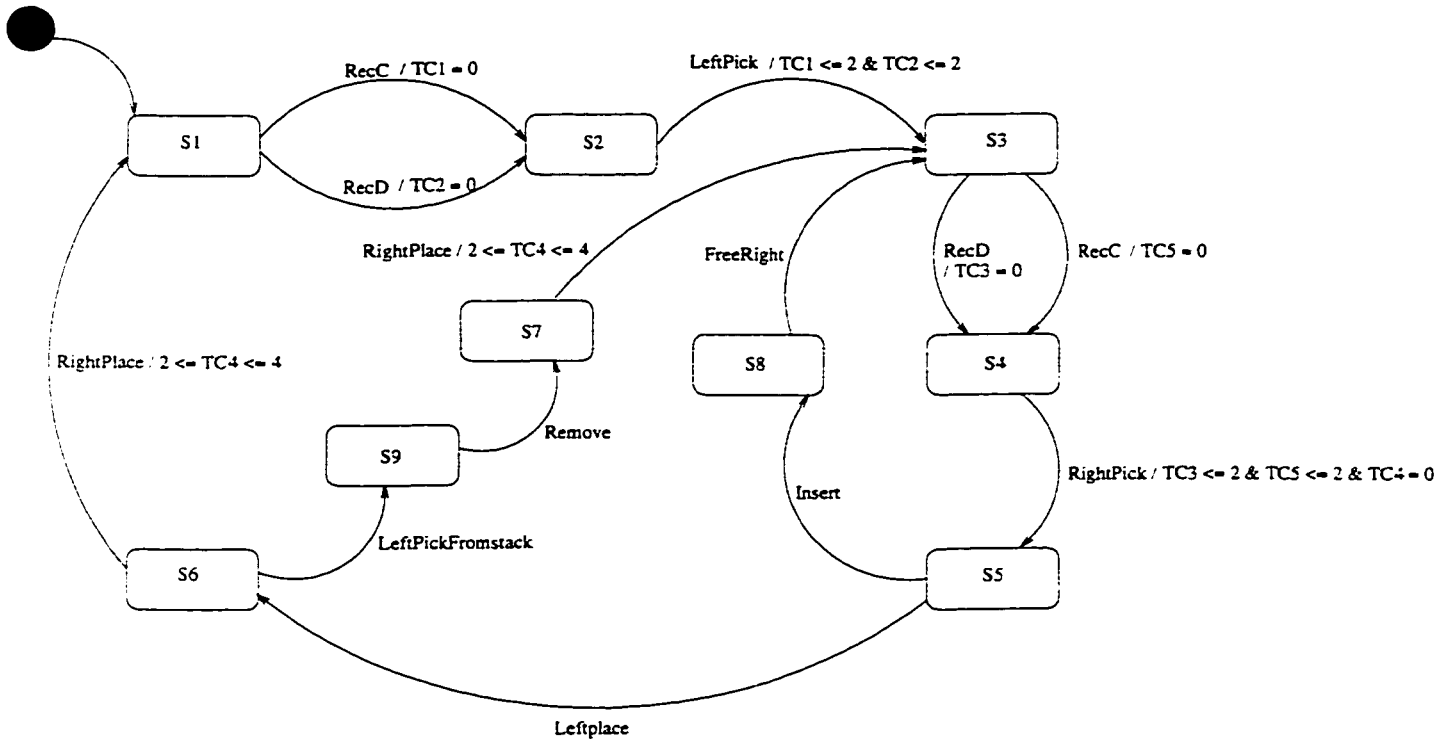
Figure 40: Belt TROM class - UML model

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. Figure 41, Figure 42, and Figure 43 show the textual representation, state machine representation, and the UML model of the *Robot* class.

```
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 41: 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 42: Robot TROM class - State machine representation

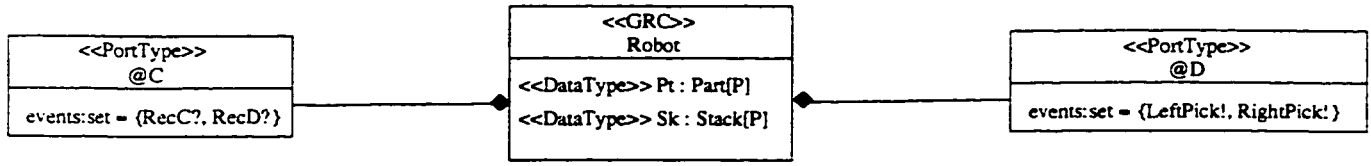


Figure 43: Robot TROM class - UML model

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 44, and Figure 45 show the textual representation, and 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 44: SCS - Textual representation

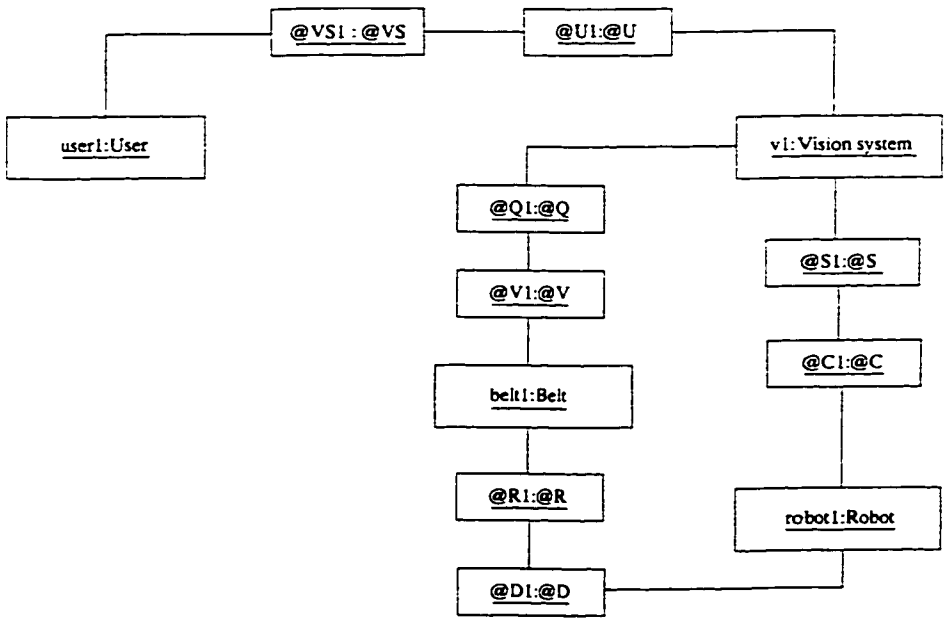


Figure 45: SCS - UML model

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 46 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 46: Sample Simulation Event List

LSL Traits

The system uses three LSL traits namely *Part*, *Queue*, and *Stack*. The *Part* is used by the *Vision system* and the *Robot*. The *Queue* trait as mentioned earlier would be used by the *Vision system* to store the parts placed on the *Belt*. The *Stack* is used by the *Robot* to push and pop the parts as mentioned in the previous section. The following figures shows the textual representation of three traits namely *Part*, *Queue* and *Stack* respectively. The Figure 47, Figure 48, and Figure 49 show the three LSL traits namely *Part*, *Queue*, and *Stack*.

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

Figure 47: 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 48: 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 49: Stack LSL Trait

6.3 *GUI* for *Robotic Assembly*

In this section, we give the window snapshots of *Graphical User Interface* in the process of designing the *Robotics Assembly* system. As mentioned in the previous chapter, the *GUI* has a main window through which separate windows for *Interpreter*, *Simulator*, *Debugger*, *Query Handler*, *Trace Analyser*, and *Reasoning System* can be opened. Through the main window the user can access the *Rose-GRC Translator* to compose the visual models of *Robotics assembly*. This interface is operational if both the *GUI* and the *Rose-GRC Translator* operate in the same platform (either UNIX or Windows). The *Rose-GRC Translator* produces the textual specifications(see Section 6.2). The files containing the textual specification then become available to the *GUI*. The user can invoke the *Interpreter GUI* at this stage.

6.3.1 *Interpreter GUI*

Figure 50 shows the *Interpreter GUI* window for type checking *Robot class* specification. The left panel in the window shows the directory tree structure for the different files in the *Robotics Assembly* system. When the user clicks the *Parse button*, the file in the text editor panel is passed to the parser. Error messages if any appear at the

bottom panel of the window. If there are errors the user can switch to the *Rose-GRC Translator* to make corrections on the visual models. The window shows the TROM class specifications for *Robotics Assembly* system in the text editor panel, and the parse messages are displayed in the bottom panel.

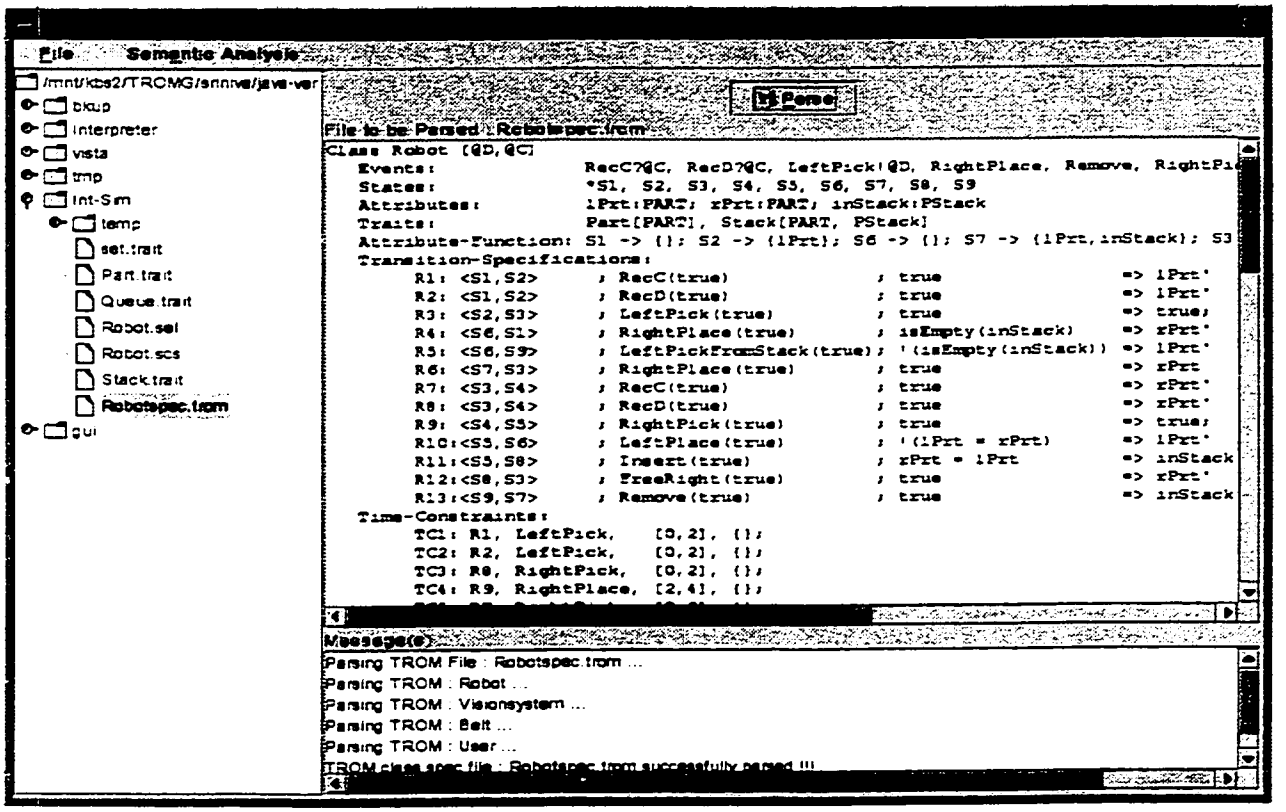


Figure 50: Window of *Interpreter GUI*

6.3.2 *Simulator GUI*

Before the user can start the simulation of the designed system, simulation parameters like *Debug option*, *Pace option* and *Timeout value* should be provided. In order to do that, the user can make use of the radio buttons to set the debug option and pace option, and the textbox is used to provide the simulation time out value. In the *Robotics Assembly* example, the *Debug option* is set to a value "yes", *Pace option* is set to a value "Normal", and the *Time out* is set to a value "20". Only after providing these values the *Start button* will be enabled to start the simulation.

Once the user clicks the *Start button*, the simulator first schedules all the *unconstrained internal events* from the initial state of respective TROM objects in the system followed by scheduling of all the simulation events in the initial simulation event list given by the user, and the corresponding rendezvous. Then the simulation checks whether there are any events to be handled at that current simulation. The simulation clock is incremented after handling any simulation events. The simulation can be viewed graphically by the user in a tabular form which is in the *Simulator GUI*. The table cell represents the state of the object in the system for a handled simulation event. A row is added to the table whenever a simulation event is handled. The simulation algorithm is given in Appendix B.

There are four objects in the *Robotics Assembly* system; in Figure 51 the first column in the table represents a *Robot* object, the second column in the table represents a *Belt* object, the third column in the table represents a *User* object, and the fourth column in the table represents a *Vision system* object. When the simulation is started, the *unconstrained internal event Next* of TROM class *User* is scheduled, followed by the simulation events in the initial Simulation Event List provided by the user. The first event to be handled by the simulator is the *output unconstrained event PutC* of TROM class *User* at simulation time 3. As soon as the simulation event is handled a row is added to the table displaying the state of each objects in the system. Then the clock is incremented, and the simulation continues until there are no simulation events in the simulation event list. Figure 51 shows the progress of simulation until time 6.

If the simulation is started in debug mode, the user will be able to invoke *Debugger GUI*, *Query Handler GUI*, *Trace Analyser GUI*, and *Reasoning System GUI* by clicking the respective menu item under the *Show Window menu*. In debug mode, the simulation clock is freezed automatically after an event is handled and the user is allowed to query the system. The simulation can be resumed by clicking the *Start button*. Figure 52, Figure 53, Figure 54, and Figure 55 show the snapshots of *Debugger GUI*, *Query Handler GUI*, *Trace Analyser GUI*, and *Reasoning System GUI* respectively.

Debugger provides a set of queries which allow the user to get information about the current status of the simulation. Following are the some of the queries in the *Debugger.*

- Display system status,
- Display subsystem status,
- Display TROM status, and
- Display simulation event list.

For example, the query *Display system status* displays the current status of *Robotics Assembly* system. The status of *Robotics Assembly* system includes the display of the current state of every TROM object, the assignment vector, and the reaction vector. *Debugger GUI* provides the set of queries in the *Debugger* to user and allows the user to select a particular query by clicking the corresponding *radio button.* After choosing a particular query the user can compose the query by giving its parameters through the text boxes provided for that query. For example, the query *show subsystem status* requires the user to provide a *subsystem name* through the text box provided for this query. Once user finishes with composing the query, the response to the query can be viewed in the bottom panel by clicking the *Show button.*

Query handler provides a set of queries which allow the user to get static information of the system. Following are the some of the queries in the *Query handler.*

- Display Trom AST,
- Display transitions for given Trom,
- Display transitions to given state, and
- Display time constraints for given Trom.

For example, the query *Display Trom AST for a given Trom object* displays the AST of the given TROM object. *Query handler GUI* provides the set of queries in the *Query handler* to user and allows the user to select a particular query by clicking the corresponding *radio button.* After choosing a particular query the user can compose the query by giving its parameters through the text boxes provided for that query.

For example, the query *Display transitions for given Trom* requires the user to provide a TROM object name through the text box provided for the query. Once user finishes with composing the query, the response to the query can be viewed in the bottom panel by clicking the *Show button.*

Trace analyser provides a set of queries which allow the user to get information about the history of the simulation. Following are the some of the queries in the *Trace analyser.*

- Display simulation events causing transition,
- Display simulation events causing no transition,
- Display simulation events not yet handled, and
- Display simulation events for given period.

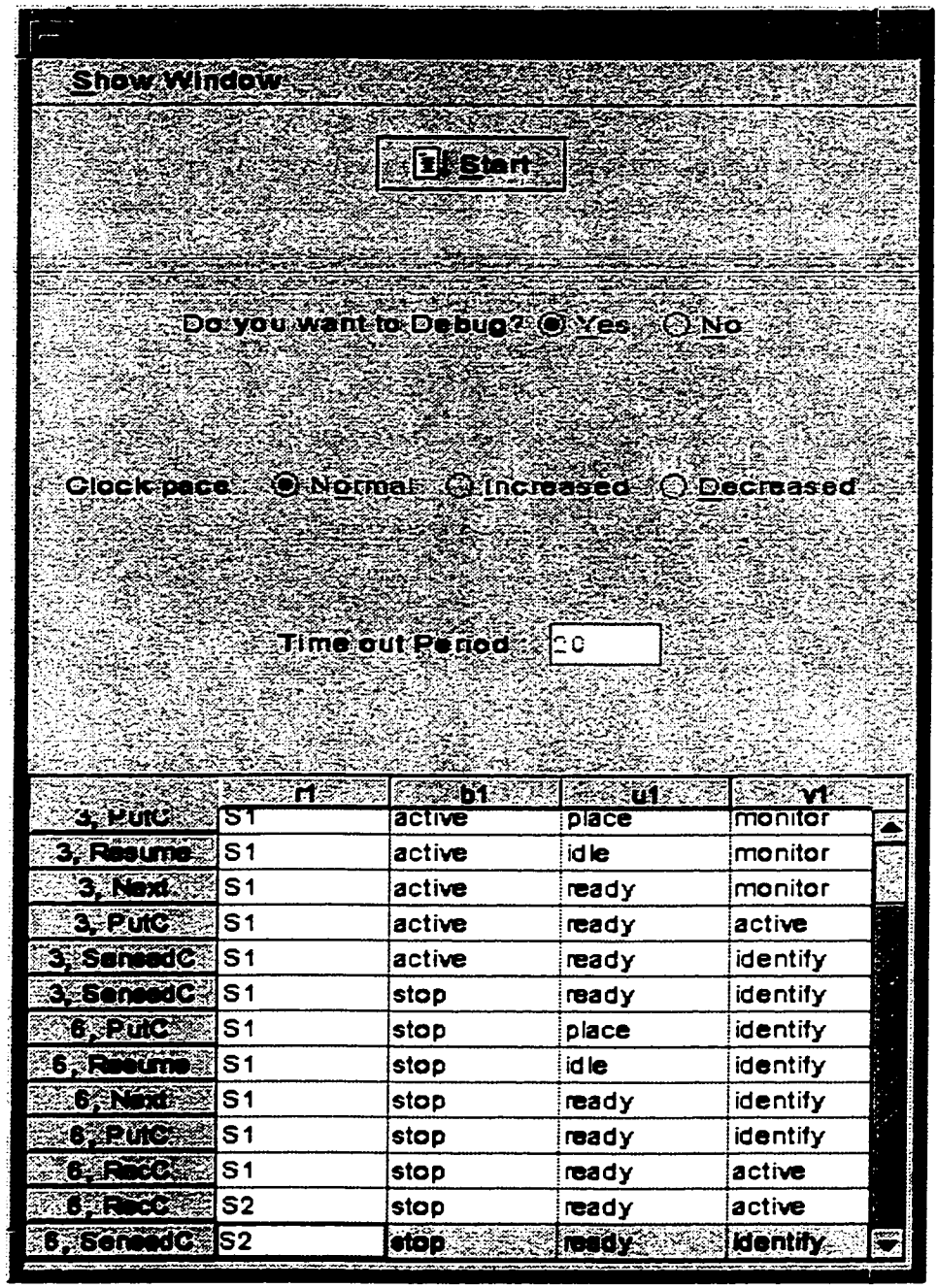
For example the query *Display simulation events causing transition,* displays the simulation events which were *enabled. Trace analyser GUI* provides the set of queries in the *Trace analyser* to user and allows the user to select a particular query by clicking the corresponding *radio button.* After choosing a particular query the user can compose the query by giving its parameters through the text boxes provided for that query. For example, the query *Display simulation events for given period* requires the user to provide a time period through the text boxes provided for the query. Once user finishes with composing the query, the response to the query can be viewed in the bottom panel by clicking the *Show button.*

Reasoning system provides a set of queries which allow the user to get the following types of information:

- History queries: The query *Why the system goes from one state to another state* is an example of a history query. The response to the query allows the user to understand the reasons which caused the transitions that lead the system go from one state to another state.
- Hypothetical queries: The query *What if we insert an event* is an example of a hypothetical query. The response to the query allows the user to analyse the consequences of inserting a new simulation event.

- Reachability queries: The query *Show all the routes between any two states of a TROM* is an example of a reachability query. The response to this query is a display of all possible acyclic routes between any two given states of a TROM object.

Reasoning system GUI provides the set of queries in the *Reasoning system* to user and allows the user to select a particular query by clicking the corresponding *radio button.* After choosing a particular query the user can compose the query by giving its parameters through the text boxes provided for that query. For example, the query *Why the system goes from one state to another state* requires the user to provide the TROM object name, and a set of states through the text boxes provided for the query. Once user finishes with composing the query, the response to the query can be viewed in the bottom panel by clicking the *Show button.*



	r1	b1	u1	v1
3, PutC	S1	active	place	monitor
3, Resume	S1	active	idle	monitor
3, Next	S1	active	ready	monitor
3, PutC	S1	active	ready	active
3, SenseC	S1	active	ready	identify
3, SenseC	S1	stop	ready	identify
6, PutC	S1	stop	place	identify
6, Resume	S1	stop	idle	identify
6, Next	S1	stop	ready	identify
6, PutC	S1	stop	ready	identify
6, RecC	S1	stop	ready	active
6, RecC	S2	stop	ready	active
6, SenseC	S2	stop	ready	identify

Figure 51: Window of *Simulator GUI*

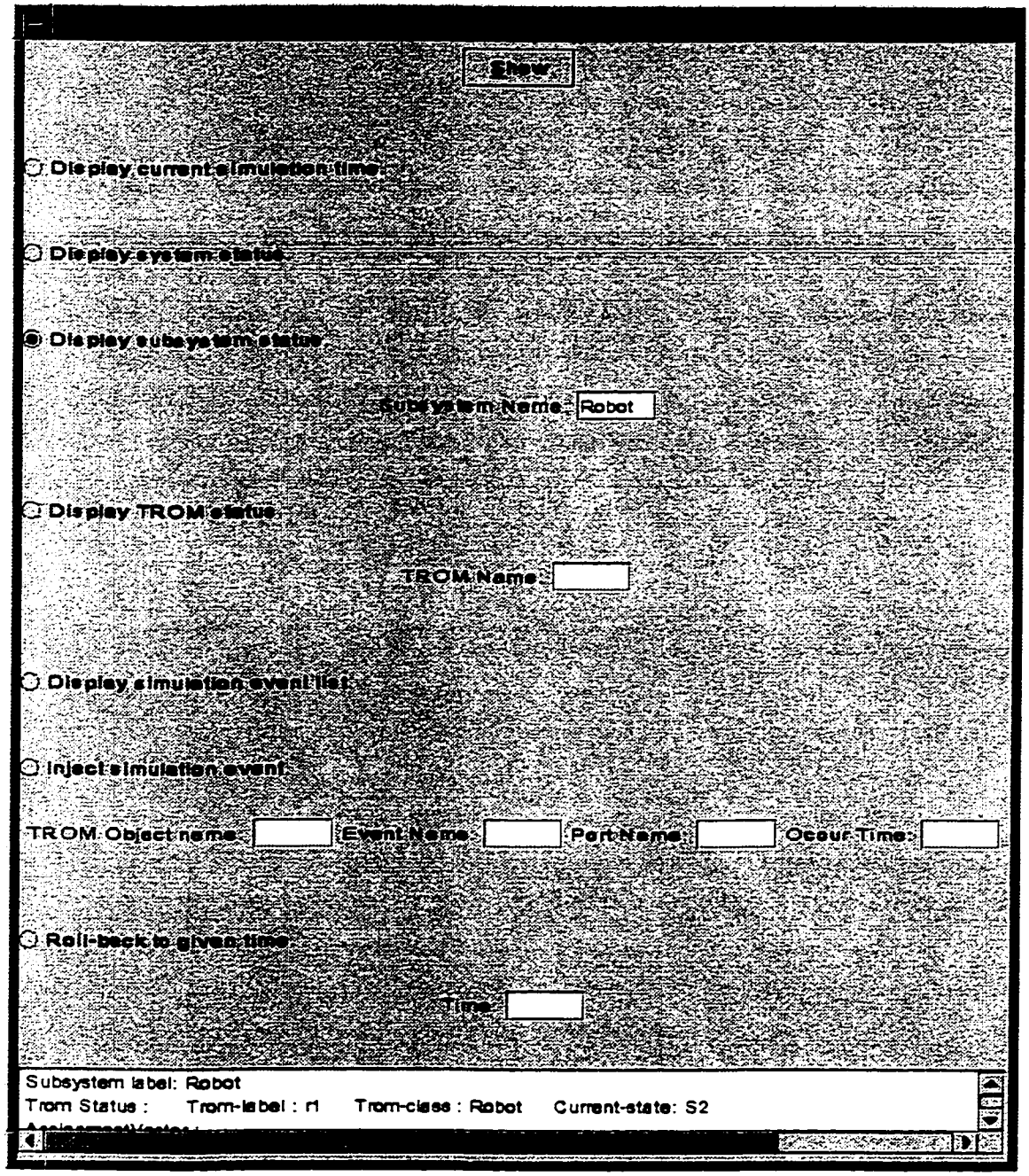


Figure 52: Window of *Simulator - Debugger GUI*

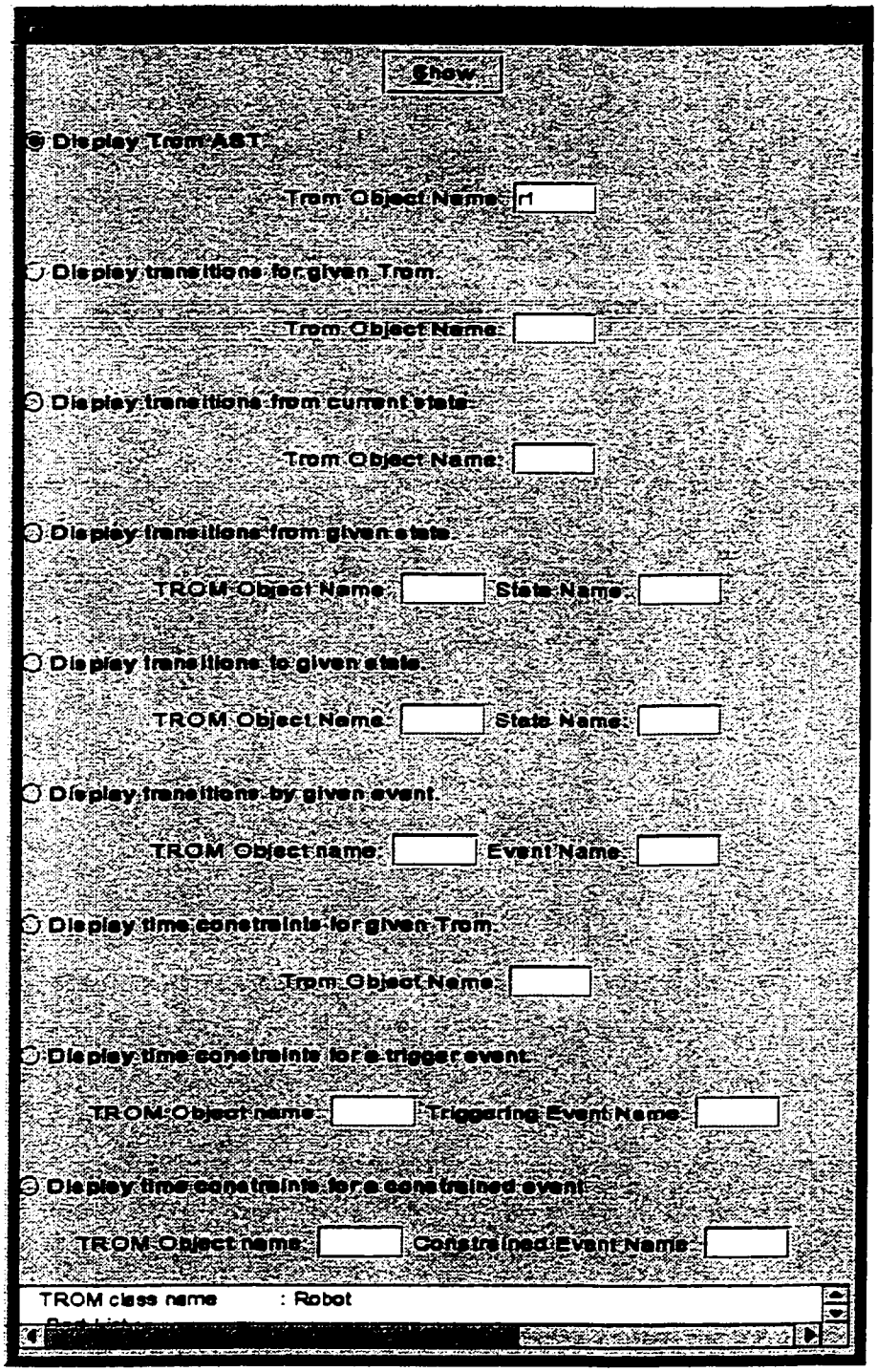


Figure 53: Window of *Simulator - Query Handler GUI*

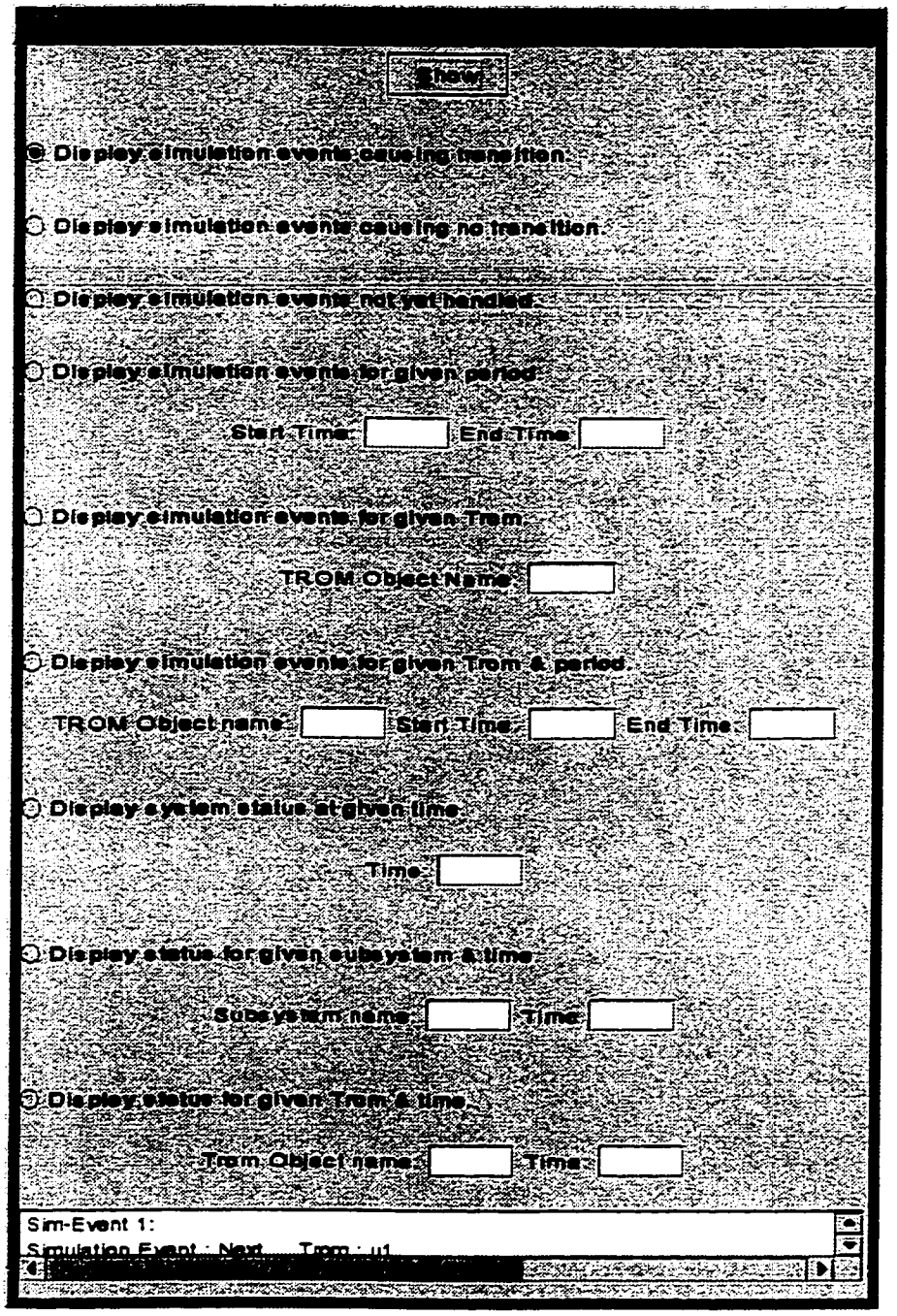


Figure 54: Window of *Simulator - Trace Analyser GUI*

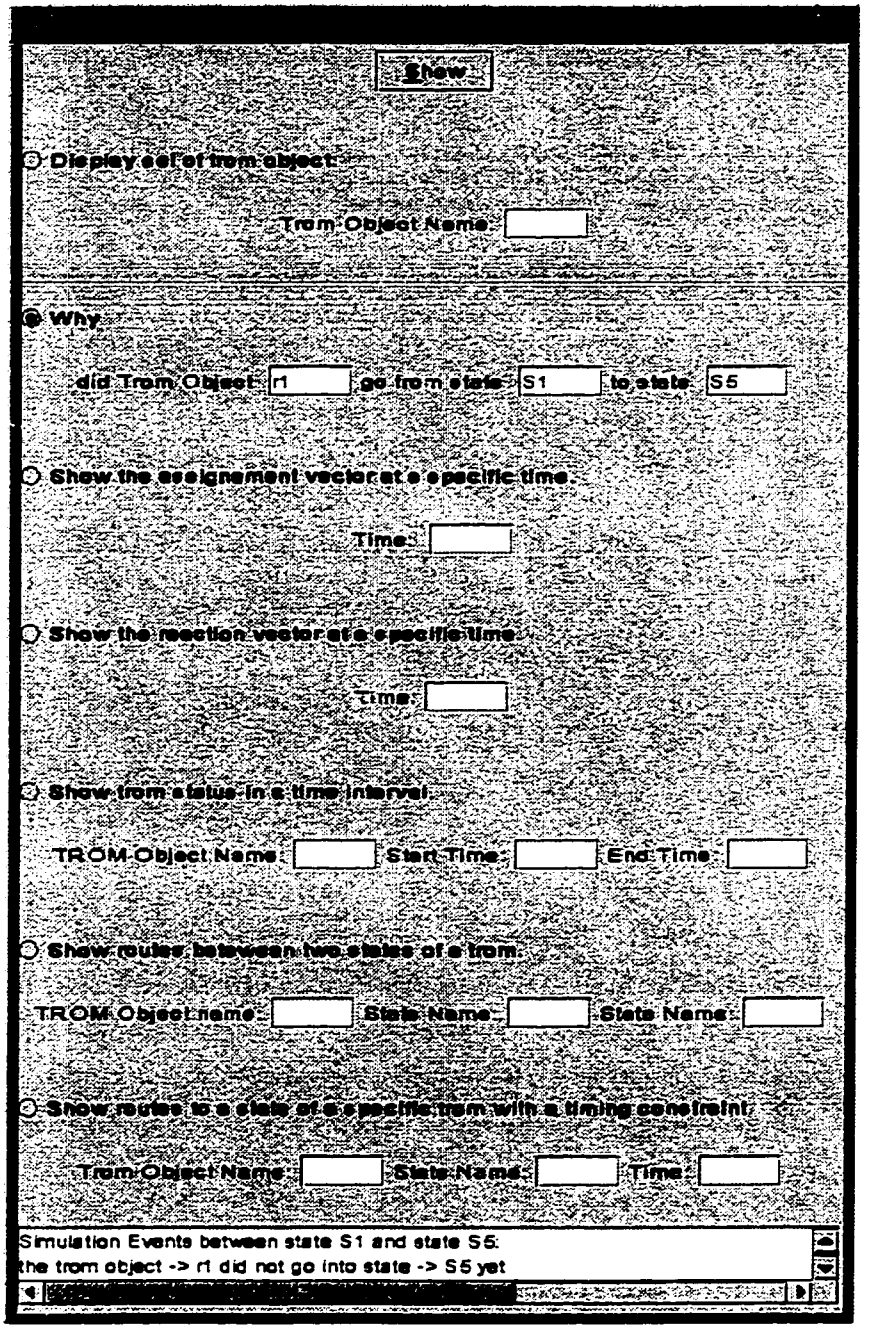


Figure 55: Window of *Simulator - Reasoning System GUI*

Chapter 7

Conclusion and Future Work

The thesis has contributed two significant results to TROMLAB:

- reengineered components which are adaptable, scalable, and reusable;
- *GUI*, which provides task-oriented interface to access all TROMLAB components

This thesis fills an important void that existed in TROMLAB. The current implementation of *GUI* can be easily extended to accomodate interfaces to any new future TROMLAB component. The users of TROMLAB have the grounds for a completely mechanically assisted prototype development cycle for a rigorous development of real-time reactive systems.

7.1 Future work

The following are some suggested future improvements to the TROMLAB environment:

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*.

2. *Simulator*

   (a) A library consisting of the implementation of a large number of LSL trait functions could be added to 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.

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

3. *GUI*

   As we saw through the previous chapter explaining the design of *GUI* and through the case study, one of the current challenges is to provide the user a more complete TROM system designing tool which allows user to draw the state machines of TROM class, increasing the usability in designing a TROM system.

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.

[Ell90] Elliot J. Chikofsky and James H. Cross II Reverse Engineering and Design Recovery: A Taxonomy in *IEEE Software Engineering*, January 1990.

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

[Hai99] G. Haidar. Simulated reasoning and debugging of **TROMLAB** environment. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, December 1999.

[Har] D. Harel, H. Lachover, A, Naamad, A. Pnueli, M. Politi, R. Sherman, A. HtullTrauring, M. Trakhtenbrot, STATEMATE: A Working Environment

for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering.*

[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.

[Jpr] Jenny Preece, Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland, and Tom Carey. Human-Computer Interaction Addison-Wesley, 1995.

[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.

[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, September 1999.

[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.

[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 */
```

Appendix C

Interface Specification in VDM

.1 *Interpreter GUI*

1. Parse Button

module *PARSE-BUTTON*

exports all

definitions

types

1.0 $SUCCESS = \textbf{token};$

2.0 $ERROR = \textbf{token};$

3.0 $Message = SUCCESS \mid ERROR;$

4.0 $String = \textbf{char}^*$

functions

5.0 $isValidSpec : String \rightarrow \textbf{B};$

6.0 $ParseLSL : String^* \rightarrow \textbf{B};$

7.0 $ParseTROM : String^* \rightarrow \textbf{B};$

8.0 $ParseSCS : String^* \rightarrow \textbf{B};$

9.0 $ParseSEL : String^* \rightarrow \mathbf{B}$;

operations

10.0 *parse-button-action* (*Specfile* : *String**, *Spectype* : *String*) *Report* : *Message*

.1 **ext wr** *isParsedLSL* : **B**

.2 **wr** *isParsedTROM* : **B**

.3 **wr** *isParsedSCS* : **B**

.4 **wr** *isParsedSEL* : **B**

.5 **pre** *Parse-button* $\in$ **dom** *RadioButtons* $\wedge$ *Simulatorwindow* $\in$ **dom** *Windows* $\wedge$

.6 (*Spectype* = "*LSL*" $\vee$ *Spectype* = "*TROM*" $\vee$ *Spectype* = "*SCS*" $\vee$ *Spectype* = "*SEL*")

.7 **post if** (*Spectype* = "*LSL*")

.8 **then** *isParsedLSL* = *ParseLSL* (*Specfile*) $\wedge$

.9 **if** (*isParsedLSL*)

.10 **then** *Report* = *SUCCESS*

.11 **else** *Report* = *ERROR*

.12 **elseif** (*Specfile* = "*TROM*")

.13 **then** *isParsedTROM* = *ParseTROM* (*Specfile*) $\wedge$

.14 **if** (*isParsedTROM*)

.15 **then** *Report* = *SUCCESS*

.16 **else** *Report* = *ERROR*

.17 **elseif** (*Specfile* = "*SCS*")

.18 **then** *isParsedSCS* = *ParseSCS* (*Specfile*) $\wedge$

.19 **if** (*isParsedSCS*)

.20 **then** *Report* = *SUCCESS*

.21 **else** *Report* = *ERROR*

.22 **else** *isParsedSEL* = *ParseSEL* (*Specfile*) $\wedge$

.23 **if** (*isParsedSEL*)

.24 **then** *Report* = *SUCCESS*

.25 **else** *Report* = *ERROR*

.26 **errs** *INVALID-SPEC-TYPE* : $\neg$ (*Spectype* = "*LSL*" $\vee$ *Spectype* = "*TROM*" $\vee$ *Spectype* = "*SCS*" $\vee$ *Spectype* = "*SEL*") $\rightarrow$ *Report* = *ERROR*

end *PARSE-BUTTON*

2. Semantic Verify Menu Item

module *SEMANTIC-VERIFY-MENU*

exports all

definitions

types

11.0 *SUCCESS* = **token**;

12.0 *ERROR* = **token**;

13.0 *Message* = *SUCCESS* | *ERROR*;

14.0 *String* = **char***;

15.0 *Specs* = **compose** *Spec* **of**
.1 *LSLSpec* : *String**
.2 *TROMSpec* : *String**
.3 *SCSSpec* : *String**
.4 *SELSpec* : *String**
.5 **end**;

16.0 *Result* = **B**

functions

17.0 *Semanticverify* : *Specs* $\rightarrow$ **B**;

operations

18.0 *sematic-verify-menu-action* () *Report* : *Message*

.1 **ext rd** *specs* : *Specs*

.2 **rd** *isParsedLSL* : **B**

.3 **rd** *isParsedTROM* : **B**

.4 **rd** *isParsedSCS* : **B**

.5 **rd** *isParsedSEL* : **B**

.6 **pre** *Semantic-verify-menu-item* ∈ **dom** *Menus* ∧ *Simulatorwindow* ∈ **dom** *Windows* ∧

.7 *isParsedLSL* ∧ *isParsedTROM* ∧ *isParsedSCS* ∧ *isParsedSEL*

.8 **post** *Result* = *Semanticverify* (*specs*) ∧

.9 **if** (*Result* = **true**)

.10 **then** *Report* = *SUCCESS*

.11 **else** *Report* = *ERROR*

end *SEMANTIC-VERIFY-MENU*

.2 *Simulator GUI*

1. Debug Radio Button

module *DEBUG-R-BUTTON*

exports all

definitions

types

19.0 *String* = **char***;

20.0 *YesLabel* = **token**;

21.0 *NoLabel* = **token**;

22.0 *Label* = *YesLabel* | *NoLabel*;

23.0 *DebugRbutton* = *Label*-**set**;

24.0 $Yes = String$;

25.0 $No = String$;

26.0 $choice = Yes \mid No$;

27.0 $Debugoption = choice\text{-}\mathbf{set}$;

28.0 $RButtonMap = DebugRbutton \xrightarrow{m} Debugoption$

operations

29.0 *debug-button-action* ()

.1 **ext wr** $Debugvalue : String$

.2 **pre** $Debug\text{-}radio\text{-}button \in \mathbf{dom}\ RadioButtons$

.3 **post** $Debugvalue = \mathbf{rng}\ (\{YesLabel\} \lhd RButtonMap) \lor \mathbf{rng}\ (\{NoLabel\} \lhd RButtonMap)$

end *DEBUG-R-BUTTON*

2. Pace Radio Button

module *PACE-R-BUTTON*

exports all

definitions

types

30.0 $String = \mathbf{char}^*$;

31.0 $NormalLabel = \mathbf{token}$;

32.0 $DecreasedLabel = \mathbf{token}$;

33.0 $IncreasedLabel = \mathbf{token}$;

34.0 $Label = NormalLabel \mid DecreasedLabel \mid IncreasedLabel$;

35.0 $PaceRbutton = Label\text{-}\mathbf{set}$;

36.0 $Normal = String$;

37.0 $Decreased = String$;

38.0 $Increased = String$;

39.0 $Choice = Normal \mid Decreased \mid Increased$;

40.0 $Paceoption = Choice\text{-}\mathbf{set}$;

41.0 $RButtonMap = PaceRbutton \xrightarrow{m} Paceoption$

operations

42.0 *pace-button-action* ()

.1 **ext wr** $Pacevalue : String$

.2 **pre** $Pace\text{-}radio\text{-}button \in \mathbf{dom}\ RadioButtons$

.3 **post** $Pacevalue = \mathbf{rng}\ (\{NormalLabel\} \lhd RButtonMap) \lor \mathbf{rng}\ (\{DecreasedLabel\} \lhd RButtonMap) \lor \mathbf{rng}\ (\{IncreasedLaebl\} \lhd RButtonMap)$

end *PACE-R-BUTTON*

3. Time Out text field

module *TIMEOUT-TEXTFIELD*

exports all

definitions

types

43.0 $SUCCESS = \mathbf{token}$;

44.0 $ERROR = \mathbf{token}$;

45.0 $Message = SUCCESS \mid ERROR$

operations

46.0 *timeout-textfield-action* (*Timeout* : $\mathbf{Z}$) *Report* : *Message*

.1 **ext wr** *Timeoutvalue* : $\mathbf{N}$

.2 **pre** $Timeout > 0$

.3 **post** $Timeoutvalue = Timeout \wedge Report = SUCCESS$

.4 **errs** $INVALID\text{-}VALUE : (Timeout \leq 0) \rightarrow (Report = ERROR)$

end *TIMEOUT-TEXTFIELD*

4. Start Button

module *START-BUTTON*

exports all

definitions

types

47.0 $Realclock = \mathbf{N}$;

48.0 $Simclock = \mathbf{N}$

functions

49.0 $Clock : Realclock \rightarrow Simclock$;

operations

50.0 *start-button-action* ($t : \mathbf{N}$)

.1 **ext rd** *Debugvalue* : *String*

.2 **rd** *Pacevalue* : *String*

.3 **rd** *Timeoutvalue* : $\mathbf{N}$

.4 **pre** $StartButton \in \mathbf{dom}\ Buttons \wedge \mathbf{len}\ (Debugvalue) \neq 0 \wedge$

.5 $\mathbf{len}\ (Pacevalue) \neq 0 \wedge Timeoutvalue \geq 0 \wedge (Clock\,(t) = Clock\,(t+1))$

.6 **post** $Clock\,(t) < Clock\,(t+1)$

end *START-BUTTON*

5. Simulation Table

module *SIMULATION-TABLE*

exports all

definitions

types

51.0 $String = \mathbf{char}^*$;

52.0 $Objectname = String^*$;

53.0 $Objectstatus = String^*$;

54.0 $Row = Cell^*$;

55.0 $Table = Row^*$

functions

56.0 $Cell : Objectname \rightarrow Objectstatus$;

57.0 $TableMap : Simclock \rightarrow Table$;

operations

58.0 *simulation-table-action* ()

.1 **ext rd** $Simclock : \mathbf{N}$

.2 **pre len** $(Objectname) \neq 0$

.3 **post len** $(TableMap\,(Simclock)) \geq$ **len** $(TableMap\,(Simclock - 1))$

end *SIMULATION-TABLE*

6. Show Histoy Menu Item

module *SHOW-HISTORY*

exports all

definitions

types

59.0 $String = \mathbf{char}^*$;

60.0 $Realclock = \mathbf{N}$;

61.0 $Simclock = \mathbf{N}$;

62.0 $HistoryMenuLabel = \mathbf{token}$;

63.0 $HistoryMenuItem = HistoryMenuLabel\text{-}\mathbf{set}$

functions

64.0 $Showhistory : HistoryMenuItem \rightarrow String$;

65.0 $Clock : Realclock \rightarrow Simclock$;

operations

66.0 $show\text{-}history\text{-}menu\text{-}action\ (t : \mathbf{N})$

.1 **ext wr** $Textareavalue : String$

.2 **pre** $Clock\ (t) = Clock\ (t + 1)$

.3 **post** $Textareavalue = Showhistory\ (HistoryMenuLabel)$

end *SHOW-HISTORY*

7. Show debugger Menu item

module *SHOW-DEBUGGER*

exports all

definitions

types

67.0 $Realclock = \mathbf{N}$;

68.0 $Simclock = \mathbf{N}$

functions

69.0 $Clock : Realclock \rightarrow Simclock$;

operations

70.0 *show-debugger-window* $(t : \mathbf{N})$

.1 **pre** $Show\text{-}debug\text{-}menu \in \mathbf{dom}\ Menus \wedge Clock\,(t) = Clock\,(t+1)$

.2 **post** $Debuggerwindow \in \mathbf{dom}\ Windows$

end *SHOW-DEBUGGER*

8. Show system status Button

module *SHOW-SYSTEM-STATUS*

exports all

definitions

types

71.0 $String = \mathbf{char}^{\bullet}$;

72.0 $Systemstatuslabel = \mathbf{token}$

functions

73.0 $Showstatus : String \rightarrow String$;

operations

74.0 *show-system-status-button* ()

.1 **ext wr** *Status-bar-value* : *String*

.2 **rd** *Currentsystem* : *String*

.3 **pre** *Show-system-status-button* $\in$ **dom** *Buttons* $\wedge$ *Debuggerwindow* $\in$ **dom** *Windows*

.4 **post** *Status-bar-value* = *Showstatus* (*Currentsystem*)

end *SHOW-SYSTEM-STATUS*

9. Show subsystem status Button

module *SHOW-SUBSYSTEM-STATUS*

exports all

definitions

types

75.0 *String* = **char***;

76.0 *SUCCESS* = **token**;

77.0 *ERROR* = **token**;

78.0 *Message* = *SUCCESS* | *ERROR*

functions

79.0 *ShowSubsystemStatus* : *String* $\times$ **N** $\rightarrow$ *String*;

operations

80.0 *show-subsystem-status-action* (*Subsystemname* : *String*, *Time* : **N**) *Report* : *Message*

.1 **ext wr** *Statusbar-value* : *String*

.2 **pre len** (*Subsystemname*) $> 0 \wedge$ *Subsystemname* $\in$ **dom** *Subsystem* $\wedge$

.3 *Show-subsystem-status-button* $\in$ **dom** *Buttons* $\wedge$

.4 *Debuggerwindow* $\in$ **dom** *Windows*

.5 **post** *Statusbar-value* = *ShowSubsystemStatus* (*Subsystemname*, *Time*) $\wedge$ *Report* = *SUCCESS*

.6 **errs** *INVALIDSUBSYTEM* : (*Subsystemname* $\notin$ **dom** *Subsystems*) $\rightarrow$ *Report* = *ERROR*

end *SHOW-SUBSYSTEM-STATUS*

10. Show TROM status Button

module *SHOW-TROM-STATUS*

exports all

definitions

types

81.0 *String* = **char***;

82.0 *SUCCESS* = **token**;

83.0 *ERROR* = **token**;

84.0 *Message* = *SUCCESS* | *ERROR*

functions

85.0 *ShowTROMstatus* : *String* $\times$ **N** $\rightarrow$ *String*;

operations

86.0 *show-TROM-staus-action* (*TROMname*:*String*, *Time*:**N**) *Report*:*Message*

.1 **ext wr** *Statusbarvalue* : *String*

.2 **pre len** (*TROMname*) $> 0 \wedge$ *TROMname* $\in$ **dom** *TROMS* $\wedge$

.3 *Show-TROM-status-button* $\in$ **dom** *Buttons* $\wedge$ *Debuggerwindow* $\in$ **dom** *Windows*

.4 **post** *Statusbar* = *ShowTROMstatus* (*TROMname*, *Time*)$\wedge$*Report* = *SUCCESS*

.5 **errs** *INVALID-TROM* : (*TROMname* $\notin$ **dom** *TROMS*) $\rightarrow$ *Report* = *ERROR*

end *SHOW-TROM-STATUS*

11. Show Simulation Event List Button

module *SHOW-SEL*

exports all

definitions

types

87.0 $String = \mathbf{char}^*$

functions

88.0 $ShowSEL : String \rightarrow String;$

operations

89.0 *show-SEL-button-action* (*SELLabel* : *String*)

.1 **ext wr** *Textareavalue* : *string*

.2 **pre** *show-SEL-Button* $\in$ **dom** *Buttons* $\wedge$ *Debuggerwindow* $\in$ **dom** *Windows*

.3 **post** *Textareavalue* = *ShowSEL* (*SELLabel*)

end *SHOW-SEL*

12. Inject a Simulation Event Button

module *INJECT-SIM-EV*

exports all

definitions

types

90.0 $ERROR = \mathbf{token};$

91.0 $SUCCESS = \mathbf{token};$

92.0 $Message = SUCCESS \mid ERROR;$

93.0 $String = \mathbf{char}^*;$

94.0 $SimEv =$ **compose** *SimEvent* **of**

.1 *TROMname* : *String*

.2 *Eventname* : *String*

.3 *Portname* : *String*

.4 *Occurtime* : **N**

.5 **end**

operations

95.0 *inject-sim-eve-action* (*Tname*:*String*, *eventname*:*String*, *portname*:*String*, *occurtime*: **N**) *Report* : *Message*

.1 **ext wr** *SEList* : *SimEvent**

.2 **rd** *Simclock* : **N**

.3 **pre** *Inject-sime-ev-button* $\in$ **dom** *Buttons*$\wedge$*Debuggerwindow* $\in$ **dom** *Windows*$\wedge$ *Tname* $\in$ **dom** *TROMS* $\wedge$

.4 *eventname* $\in$ **dom** *Events* $\wedge$ *portname* $\in$ **dom** *Ports* $\wedge$ *occurtime* $>$ *Simclock*

.5 **post** *SEList* $=$ *SEList*$^\frown$[**mk-***SimEvent* (*Tname*, *eventname*, *portname*, *occurtime*)]$\wedge$ *Report* $=$ *SUCCESS*

.6 **errs** *INVALID-SIM-EVENT* : ((*Tname* $\notin$ **dom** *TROMS*)$\wedge$(*eventname* $\notin$ **dom** *Events*) $\wedge$ (*portname* $\notin$ **dom** *Ports*) $\wedge$ (*occurtime* $<$ *Simclock*)) $\rightarrow$ *Report* $=$ *ERROR*

end *INJECT-SIM-EV*

13. Rollback the Simulation to a given time Button

module *ROLLBACK*

exports all

definitions

types

96.0 $SUCCESS =$ **token**;

97.0 $ERROR =$ **token**;

98.0 $Message = SUCCESS \mid ERROR$

operations

99.0 *roll-back-action* (*Rollbacktime* : **N**) *Report* : *Message*

.1 **ext rd** *Simclock* : **N**

.2 **pre** *Rollback-button* ∈ **dom** *Buttons*∧*Debuggerwindow* ∈ **dom** *Windows*∧

.3 *Rollbacktime* ≤ *Simclock*

.4 **post** *Simclock* = *Rollbacktime*∧(∀ $x \in SEList \cdot x.Occurtime < Rollbacktime$)∧ *Report* = *SUCCESS*

.5 **errs** *INVALID-ROLLBACK-TIME* : (*Rollbacktime* > *Simclock*) → *Report* = *ERROR*

end *ROLLBACK*

14. Show Query Handler Menu item

module *SHOW-QUERY-HANDLER*

exports all

definitions

types

100.0 *Realclock* = **N**;

101.0 *Simclock* = **N**

functions

102.0 *Clock* : *Realclock* → *Simclock*;

operations

103.0 *show-queryhandler-window-action* (*t* : **N**)

.1 **ext rd** *Realclock* : **N**

.2 **rd** *Simclock* : **N**

.3 **pre** *Show-query-handler-menu-item* ∈ **dom** *Menus*∧*Clock* (*t*) = *Clock* (*t*+ 1)

.4 **post** *Query-handler-window* $\in$ **dom** *Windows*

end *SHOW-QUERY-HANDLER*

15. Show TROM transitions Button

module *SHOW-TROM-TRANSITIONS*

exports all

definitions

types

104.0 $SUCCESS = \textbf{token};$

105.0 $ERROR = \textbf{token};$

106.0 $Message = SUCCESS \mid ERROR;$

107.0 $String = \textbf{char}^*;$

108.0 $Transitions = String^*$

functions

109.0 $showtransitions : String \rightarrow Transitions;$

operations

110.0 *show-TROM-transitions* (*TROMname* : *String*) *Report* : *Message*

.1 **ext wr** *Textareavalue* : *String**

.2 **pre** *Show-TROM-transitions-button* $\in$ **dom** *Buttons* $\wedge$ *Query-handler-window* $\in$ **dom** *Windows* $\wedge$

.3 *TROMname* $\in$ **dom** *TROMS*

.4 **post** *Textareavalue* = *showtransitions* (*TROMname*) $\wedge$ *Result* = *SUCCESS*

.5 **errs** *INVALID-TROM* : (*TROMname* $\notin$ **dom** *TROMS*) $\rightarrow$ (*Report* = *ERROR*)

end *SHOW-TROM-TRANSITIONS*

16. Show Transitions for current state of a given TROM Button

module *SHOW-TROM-CURRENT-STATE-TRANSITIONS*

exports all

definitions

types

111.0 $SUCCESS = \textbf{token};$

112.0 $ERROR = \textbf{token};$

113.0 $Message = SUCCESS \mid ERROR;$

114.0 $String = \textbf{char}^{*};$

115.0 $Transitions = String^{*}$

functions

116.0 $showcurrentstatetransition : String \times String \rightarrow Transitions;$

operations

117.0 *show-TROM-current-state-transition* (*TROMname*:*String*) *Report*:*Message*

.1 **ext rd** *TROMcurrentstate* : *String*

.2 **wr** *Textareavalue* : *String**

.3 **pre** *Show-TROM-current-state-transition-button* $\in$ **dom** *Buttons* $\wedge$ *Query-handler-winc*
dom *Windows* $\wedge$

.4 *TROMname* $\in$ **dom** *TROMS*

.5 **post** *Textareavalue* = *showcurrentstatetransition* (*TROMname*, *TROMcurrentstate*) $\wedge$
Result = *SUCCESS*

.6 **errs** *INVALID-TROM* : (*TROMname* $\notin$ **dom** *TROMS*) $\rightarrow$ (*Report* = *ERROR*)

end *SHOW-TROM-CURRENT-STATE-TRANSITIONS*

17. Show transitions for a given trom for a given state Button

module *SHOW-TROM-GIVEN-STATE-TRANSITIONS*

exports all

definitions

types

118.0 $SUCCESS = \textbf{token};$

119.0 $ERROR = \textbf{token};$

120.0 $Message = SUCCESS \mid ERROR;$

121.0 $String = \textbf{char}^*;$

122.0 $Transitions = String^*$

functions

123.0 $showgivenstatetransition : String \times String \rightarrow Transition;$

operations

124.0 *show-TROM-given-state-transition* (*TROMname*:*String*, *Statename*:*String*) *Report*: *Message*

.1 **ext wr** *Textareavalue* : *String**

.2 **pre** *Show-TROM-given-state-transition-button* $\in$ **dom** *Buttons*$\wedge$*Query-handler-windo*
dom *Windows* $\wedge$

.3 *TROMname* $\in$ **dom** *TROMS*$\wedge$*Statename* $\in$ **elems** (*TROMstates* (*TROMname*))

.4 **post** *Textareavalue* = *showgivenstatetransition* (*TROMname*, *Statename*)$\wedge$
Result = *SUCCESS*

.5 **errs** *INVALID-TROM* : (*TROMname* $\notin$ **dom** *TROMS*) $\rightarrow$ (*Report* = *ERROR*)

end *SHOW-TROM-GIVEN-STATE-TRANSITIONS*

18. Show transitions for a given trom to a given state Button

module *SHOW-TROM-TO-GIVEN-STATE-TRANSITIONS*

exports all

definitions

types

125.0 $SUCCESS = \textbf{token};$

126.0 $ERROR = \textbf{token};$

127.0 $Message = SUCCESS \mid ERROR;$

128.0 $String = \textbf{char}^*;$

129.0 $Transitions = String^*$

functions

130.0 $showtogivenstatetransition : String \times String \rightarrow Transition;$

operations

131.0 *show-TROM-to-given-state-transition* (*TROMname* : *String*, *Statename* : *String*) *Report* : *Message*

.1 **ext wr** *Textareavalue* : *String**

.2 **pre** *Show-TROM-to-given-state-transition-button* $\in$ **dom** *Buttons*$\wedge$*Query-handler-win* **dom** *Windows* $\wedge$

.3 *TROMname* $\in$ **dom** *TROMS*$\wedge$*Statename* $\in$ **elems** (*TROMstates* (*TROMname*))

.4 **post** *Textareavalue* = *showtogivenstatetransition* (*TROMname*, *Statename*)$\wedge$ *Result* = *SUCCESS*

.5 **errs** *INVALID-TROM* : (*TROMname* $\notin$ **dom** *TROMS*) $\rightarrow$ (*Report* = *ERROR*)

end *SHOW-TROM-TO-GIVEN-STATE-TRANSITIONS*

19. Show transitions for a trom for a given event Button

module *SHOW-TROM-GIVEN-EVENT-TRANSITIONS*

exports all

definitions

types

132.0 $SUCCESS = \textbf{token};$

133.0 $ERROR = \textbf{token};$

134.0 $Message = SUCCESS \mid ERROR;$

135.0 $String = \textbf{char}^*;$

136.0 $Transitions = String^*$

functions

137.0 $showgiveneventtransition : String \times String \rightarrow Transitions;$

operations

138.0 *show-TROM-given-event-transition* (*TROMname*:*String*, *Eventname*:*String*) *Report*: *Message*

.1 **ext wr** *Textareavalue* : *String*

.2 **pre** *Show-TROM-given-event-transition-button* $\in$ **dom** *Buttons* $\wedge$ *Query-handler-windc* **dom** *Windows* $\wedge$

.3 *TROMname* $\in$ **dom** *TROMS* $\wedge$ *Eventname* $\in$ **elems** (*TROMevents* (*TROMname*))

.4 **post** *Textareavalue* = *showgiveneventtransition* (*TROMname*, *Eventname*) $\wedge$ *Result* = *SUCCESS*

.5 **errs** *INVALID-TROM* : (*TROMname* $\notin$ **dom** *TROMS*) $\rightarrow$ (*Report* = *ERROR*)

end *SHOW-TROM-GIVEN-EVENT-TRANSITIONS*

20. Show time constraints of a given TROM Button

module *SHOW-TROM-TIMECONSTRAINTS*

exports all

definitions

types

139.0 $SUCCESS = \textbf{token};$

140.0 $ERROR = \textbf{token};$

141.0 $Message = SUCCESS \mid ERROR;$

142.0 $String = \textbf{char}^*;$

143.0 $Timeconstraints = String^*$

functions

144.0 $showtimeconstraints : String \rightarrow Timeconstraints;$

operations

145.0 $show\text{-}TROM\text{-}timeconstraints\ (TROMname : String)\ Report : Message$

.1 **ext wr** $Textareavalue : String^*$

.2 **pre** $Show\text{-}TROM\text{-}timeconstraints\text{-}button \in \textbf{dom}\ Buttons \wedge Query\text{-}handler\text{-}window \in \textbf{dom}\ Windows \wedge$

.3 $TROMname \in \textbf{dom}\ TROMS$

.4 **post** $Textareavalue = showtimeconstraints\ (TROMname) \wedge Result = SUCCESS$

.5 **errs** $INVALID\text{-}TROM : (TROMname \notin \textbf{dom}\ TROMS) \rightarrow (Report = ERROR)$

end *SHOW-TROM-TIMECONSTRAINTS*

21. Show time constraints for a triggering event of a given TROM Button

module *SHOW-TROM-GIVEN-TRIGGERING-EVENT-TIMECONSTRAINTS*

exports all

definitions

types

146.0 $SUCCESS = \textbf{token};$

147.0 $ERROR = \textbf{token};$

148.0 $Message = SUCCESS \mid ERROR;$

149.0 $String = \textbf{char}^*;$

150.0 $TROMevents = String^*;$

151.0 $Timeconstraints = String^*$

functions

152.0 $showgiventrigevtimeconstraints : String \times String \rightarrow Timeconstraint;$

operations

153.0 *show-TROM-timeconstraints-given-triggering-event* (*TROMname*:*String*, *Eventname*: *String*) *Report* : *Message*

.1 **ext wr** *Textareavalue* : *String**

.2 **pre** *Show-TROM-given-triggering-event-timeconstraints-button* $\in$ **dom** *Buttons* $\wedge$ *Query-handler-window* $\in$ **dom** *Windows* $\wedge$

.3 *TROMname* $\in$ **dom** *TROMS* $\wedge$ *Eventname* $\in$ **elems** (*TROMevents* (*TROMname*))

.4 **post** *Textareavalue* = *showgiventrigevtimeconstraints* (*TROMname*, *Eventname*) $\wedge$ *Result* = *SUCCESS*

.5 **errs** *INVALID-TROM* : (*TROMname* $\notin$ **dom** *TROMS*) $\rightarrow$ (*Report* = *ERROR*)

end *SHOW-TROM-GIVEN-TRIGGERING-EVENT-TIMECONSTRAINTS*

22. Show timeconstraints for a constrained event of a given TROM Button

module *SHOW-TROM-GIVEN-CONSTRAINED-EVENT-TIMECONSTRAINTS*

exports all

definitions

types

154.0 $SUCCESS = \textbf{token};$

155.0 $ERROR = \textbf{token};$

156.0 $Message = SUCCESS \mid ERROR;$

157.0 $String = \textbf{char}^*;$

158.0 $TROMevents = String^*;$

159.0 $Timeconstraints = String^*$

functions

160.0 $showgivenconstevtimeconstraints : String \times String \rightarrow Timeconstraints;$

operations

161.0 *show-TROM-timeconstraints-given-constrained-event* (*TROMname*:*String*, *Eventname*: *String*) *Report* : *Message*

.1 **ext wr** *Textareavalue* : *String**

.2 **pre** *Show-TROM-given-constrained-event-timeconstraints-button* $\in$ **dom** *Buttons* $\wedge$ *Query-handler-window* $\in$ **dom** *Windows* $\wedge$

.3 *TROMname* $\in$ **dom** *TROMS* $\wedge$ *Eventname* $\in$ **elems** (*TROMevents* (*TROMname*))

.4 **post** *Textareavalue* = *showgivenconstevtimeconstraints* (*TROMname*, *Eventname*) $\wedge$ *Result* = *SUCCESS*

.5 **errs** *INVALID-TROM* : (*TROMname* $\notin$ **dom** *TROMS*) $\rightarrow$ (*Report* = *ERROR*)

end *SHOW-TROM-GIVEN-CONSTRAINED-EVENT-TIMECONSTRAINTS*

23. Show Trace Analyser Menu item

module *SHOW-TRACE-ANALYSER*

exports all

definitions

types

162.0 $Realclock = \mathbf{N}$;

163.0 $Simclock = \mathbf{N}$

functions

164.0 $Clock : Realclock \rightarrow Simclock$;

operations

165.0 *show-traceanalyser-window-action* $(t : \mathbf{N})$

.1 **ext rd** $Realclock : \mathbf{N}$

.2 **rd** $Simclock : \mathbf{N}$

.3 **pre** *Show-trace-analyser-menu-item* $\in$ **dom** $Menus \wedge Clock\,(t) = Clock\,(t+1)$

.4 **post** *Trace-analyser-window* $\in$ **dom** *Windows*

end *SHOW-TRACE-ANALYSER*

24. Show Simulation events causing the Transitions Button

module *SHOW-SIM-EVENTS-CAUSING-TRANSITIONS*

exports all

definitions

types

166.0 $String = \mathbf{char}^*$;

167.0 $SimEv = String^*$

functions

168.0 $ShowSE : String \rightarrow SimEv$;

operations

169.0 *show-simulation-events-causing-transitions-button-action* ()

.1 **ext rd** *Currenttransition* : *String*

.2 **wr** *Textareavalue* : *String**

.3 **pre** *Show-simulation-events-button* $\in$ **dom** *Buttons* $\wedge$ *Trace-analyser* $\in$ **dom** *Windows*

.4 **post** *Textareavalue* = *ShowSE* (*Currenttransition*)

end *SHOW-SIM-EVENTS-CAUSING-TRANSITIONS*

25. Show not yet handled Simulation events Button

module *SHOW-NOT-YET-HANDLED-SIM-EVENTS*

exports all

definitions

types

170.0 $String = \mathbf{char}^*$;

171.0 $SimEv = String^*$

values

172.0 $t : String = "NotYetHandled"$

functions

173.0 $ShownotyethandledSE : String \rightarrow SimEv$;

operations

174.0 *show-not-yet-handled-simulation-events-button-action* ()

.1 **ext wr** *Textareavalue* : *String**

.2 **pre** *Show-simulation-events-button* $\in$ **dom** *Buttons* $\wedge$ *Trace-analyser* $\in$ **dom** *Windows*

.3 **post** *Textareavalue* = *ShownotyethandledSE* (*t*)

end *SHOW-NOT-YET-HANDLED-SIM-EVENTS*

26. Show Simulation events for a given time period Button

module *SHOW-SIM-EVENTS-GIVEN-TIME-PERIOD*

exports all

definitions

types

175.0 *SUCCESS* = **token**;

176.0 *ERROR* = **token**;

177.0 *Message* = *SUCCESS* | *ERROR*;

178.0 *String* = **char***;

179.0 *Timep* = **compose** *Timeperiod* **of**

.1 *Low* : **N**

.2 *Up* : **N**

.3 **end**;

180.0 *SimEv* = *String**

functions

181.0 *ShowgiventpSE* : *Timep* $\rightarrow$ *SimEv*;

operations

182.0 *show-simulation-events-given-timeperiod-button-action* (*tp*:*Timep*) *Report*: *Message*

.1 **ext wr** *Textareavalue* : *String**

.2 **pre** *Show-simulation-events-button* $\in$ **dom** *Buttons* $\wedge$ *Trace-analyser* $\in$ **dom** *Windows* $\wedge$

.3 $tp.Low > 0 \wedge tp.Up > tp.Low$

.4 **post** *Textareavalue* = *ShowgiventpSE* (*tp*) $\wedge$ *Report* = *SUCCESS*

.5 **errs** *INVALID-TIME-PERIOD* : $(tp.Low < 0 \vee tp.Up < tp.Low) \rightarrow$ *Report* = *ERROR*

end *SHOW-SIM-EVENTS-GIVEN-TIME-PERIOD*

27. Show simulation events for a given TROM Button

module *SHOW-SIM-EVENTS-GIVEN-TROM*

exports all

definitions

types

183.0 *SUCCESS* = **token**;

184.0 *ERROR* = **token**;

185.0 *Message* = *SUCCESS* | *ERROR*;

186.0 *String* = **char***;

187.0 *SimEv* = *String**

functions

188.0 *ShowgivenTROMSE* : *String* $\rightarrow$ *SimEv*;

operations

189.0 *show-simulation-events-given-trom-button-action* (*TROMname*:*String*) *Report*: *Message*

.1 **ext wr** *Textareavalue* : *String**

.2 **pre** *Show-simulation-events-button* $\in$ **dom** *Buttons* $\wedge$ *Trace-analyser* $\in$ **dom** *Windows* $\wedge$

.3 $\quad$ *TROMname* $\in$ **dom** *TROMS*

.4 **post** *Textareavalue* = *ShowgiveTROMSE* (*TROMname*) $\wedge$ *Report* = *SUCCESS*

.5 **errs** *INVALID-TROM* : (*TROMname* $\notin$ **dom** *TROMS*) $\rightarrow$ *Report* = *ERROR*

end *SHOW-SIM-EVENTS-GIVEN-TROM*