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. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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.

UMI®

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

STATIC ANALYZER – A DESIGN TOOL FOR TROM

HONGJING TAO

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

AUGUST 1996

Abstract

Static Analyzer – A Design Tool for TROM

Hongjing Tao

Real-Time Reactive Systems are large complex systems. Many reserchers have been studying this field and have developed specification methods to reason about the behavior and attributes of real time reactive systems from different perspectives. This thesis contributes to the development of one part of a tool that will provide an environment to specify, design, debug and simulate real-time reactive systems built on **TROM**s, Timed Reactive Object Models. The tool consists of three major parts: Editor, Interpreter(including Axiom Generator) and Simulator. The Interpreter which is fundamental to the tool is the subject of study in this thesis. The Interpreter will do syntax and semantic analysis for user specification and generate internal data representation to perform the simulation. The Axiom Generator will generate axioms for each particular **TROM** to be used by formal verification during system simulation.

Acknowledgments

I am greatly indebted to my supervisor Professor V. S. Alagar for his valubale advice to complete this work. I would like to thank him for his continuous guidance and moral support.

I would like to thank my husband Xiaohui for his great support and encouragement.

I would also like to thank my parents for their continuous love and help.

Finally, I would like to dedicate this thesis to my lovely daughter Elizabeth.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

According to Parnas[12], a system can be considered complex if its shortest useful description is relatively long. The length of the shortest description indicates the amount of information required to understand the product. As systems become more and more sophisticated in the different domains of applications, a clean and comprehensive specification of overall system organization and behavior become critical aspects of system design. Many reserchers in this field are developing tools to deal with system complexity in different perspectives. Because the complexity in modern systems is due to software construction, it follows that we need to distinguish those attributes of software that contributes to the complexity from the physical system that they govern. Since we have to work with very complex systems in the real world, we must understand the sources of complexity and the method with which we can fight complexity.

1.1 Large Complex System

According to general system theory, an entity called "system" is a complex organization of elements or parts "in interaction"[2]. In other words, a large and complex software system contains a variety of entities(objects) and a complex system(transformation function) controlling the interaction of objects[16]. There are two types of complexities in designing a large system: *complex requirements*, and *complexity due to bad design*. In general, there are three important factors contributing

to those two types of complexities[2].

- "Largeness" in numbers: such as size(lines of code), control and distribution of operations, long life cycles and evolutionary changes, persistance of information and protection of long-term investments make the system large in space and in time.

- Heterogeneity of concepts and procedures: such as heterogeneity of equipment, operating systems heterogeneity, heterogeneity of authority, applications heterogeneity, where it is desired to integrate otherwise separate applications to perform a single task.

- Complex organization and scalability: focus on examining the relationships between the components constituting the system and identifying types of system organizations.

We should keep in mind that although the world is not simple and the system requirements become very complicated, our goal must focus on building simple systems that perform complex tasks rather than developing large complex computer systems. Since the system requirements can not be changed, the complex requirements must be understood through the use of browser, cross-references and other computerized support. The significant challenge lies in designing a system which reduces, if not totally eliminates, complexity.

There are several ways to deal with complexity. An important design principle is to reduce the complexity of interconnectedness of system components. By applying the principles of *abstraction*, *information hiding*, *separation of concerns* and *object-orientation*, system design can be simplified. For example, by structuring the system hierachy and by providing abstract interfaces, we can enhance the understandability and usability of the system. In order to achieve this goal, a modular and hierarchical design is highly recommended since it is suitable to use above mentioned method to reduce system complexity.

Recently, an Object-Oreinted methodology combining real time has been put forth to formally develop large reactive systems. The formal Object Oriented methodology introduced by Achuthan[1] includes the principles of abstraction, object orientation, separation of concerns, hierarchical design and moduler composition. The basic building block in this method is a **TROM**, Timed Reactive Object Model. Consequently, this methodology is excellent in reducing design complexity. This work provides formal foundation and rigorous methodology enabling formal verification and validation. However, in order to apply this methodology in practice, we need an environment for easy user interaction towards a better understanding of the system features, system interaction and system behavior. Our task is to build an environment for a reactive system development so that user can create a correct system specification and simulate the system behaviour. The animation tool consists of three major parts – an User Interface, an Interpreter and a Simulator. This thesis is a contribution to the development of the interpreter.

1.2 Scope of the thesis

The goal of the thesis is to develop an interpreter for syntatic and semantic checking of user defined **TROM**s and subsystems modeling a reactive system. After defining a grammar for **TROM**, we explain how it is used for syntax checking of user defined **TROM**s. Simple semantic analysis such as the correctness of event name and port-type name can be done during syntax analysis. However, issues ralated to system behavior are to be dealt with during semantic analysis. An abstract internal representation called AST, Abstract Syntax Tree, is generated for syntacticly and semanticly correct **TROM**s and Subsystems.

In Achuthan[1], an axiomatization of **TROM** is given. Based on this set of axioms, we can develop an axiom generator and provide a methodology for generating axioms for specific **TROM** models. Those axioms are exported to, a verification system, for formal verification of system properties.

1.3 Thesis Outline

A brief outline of the thesis is as follows: Chapter 2 gives a brief discussion on reactive system design: Chapter 3 discusses the architecture of the Interpreter; Chapter 4 gives grammars for **TROM**s and Subsystems. It also gives the definition of well-formed **TROM**s and subsystems and explains the syntactic analysis of user input system components; The structure of abstract syntax tree is explained and is illustrated for train-gate-controller example in Chapter 5. Chapter 6 discusses the semantic analysis and Larch/C++ specifications; axiom generator is discussed in Chapter 7. This thesis concludes in Chapter 8 with discussion on how this work fits with the overall goal of building an environment in which reactive systems based upon **TROM**s Achuthan[1] can be developed.

Chapter 2

Reactive System Design - A brief Outline

2.1 Modular Design

Real-time reactive systems are large and complex systems. Some aspects of such a system are fundamental, while others are arbitrary and are likely to change. By decomposing system functionalities into self-contained and independent modules, we can tackle the complexity arising due to system changes. Some of the advantages are the following:

- Data and functionalities can be encapsulated.
- Abstract interfaces can be provided.
- It can reduce the strength of inter-module connections. Strength of connection is consistent with an information theoretic point of view.
- Each module can be documented independent of other modules. This allows the user to focus on the small components of the system.
- Modules can be tested independently for correctness and completeness.
- Modular verification reduces the complexity of the system verification.

Hierarchical Design is another modular design concept which restructures the system and simplifies the description and analysis of the system. *Module containment hierachy, program uses hierarchy, resource control hierarchies* and *process work allocation hierarchies* are used in designing system hierarchies. Due to hierarchical design, the overall system architecture becomes easy to comprehend, thus reducing the complexity arising in the description of systems. The OO methodology introduced by Achuthan[1] includes these principles and in addition deals with real-time reactive objects.

2.2 TROM Methodology

Reactive systems[4] are systems that continually interact with their environment. Typical requirements of such systems are that they satisfy certain timing constraints and avoid unsafe execution paths during their interaction with the environment. The methodology introduced by Achuthan[1] provides a formal Object-Oriented framework for the specification and reasoning of reactive systems. In this methodology, a system requirement is specified in temporal logic[11] and a system design is modeled as a collection of synchronously communicating objects using a 3-tiered design language. See Figure 1

The middle tier gives the detailed specification of the objects used in the upper tier by means of class definitions described in **TROM** terminology. In other words, each reactive object is formally modeled using a **TROM**. A **TROM** is a finite state machine augmented with ports, attributes, timing constraints and logical assertions. The state can be hierarchical in nature. The transitions are labled by events, which describe interactions of the object with its environment. The attributes in each state, defined by the attribute function, model the computations on data associated with transitions. The behaviors of attributes are abstractly specified in the **LSL** traits included in lower tier. With each transition, three assertions are associated: 1) *precondition*, stating the conditions to be satisfied for enabling the transition; 2) *postcondition*, specifying the status of the attributes due to data computations associated

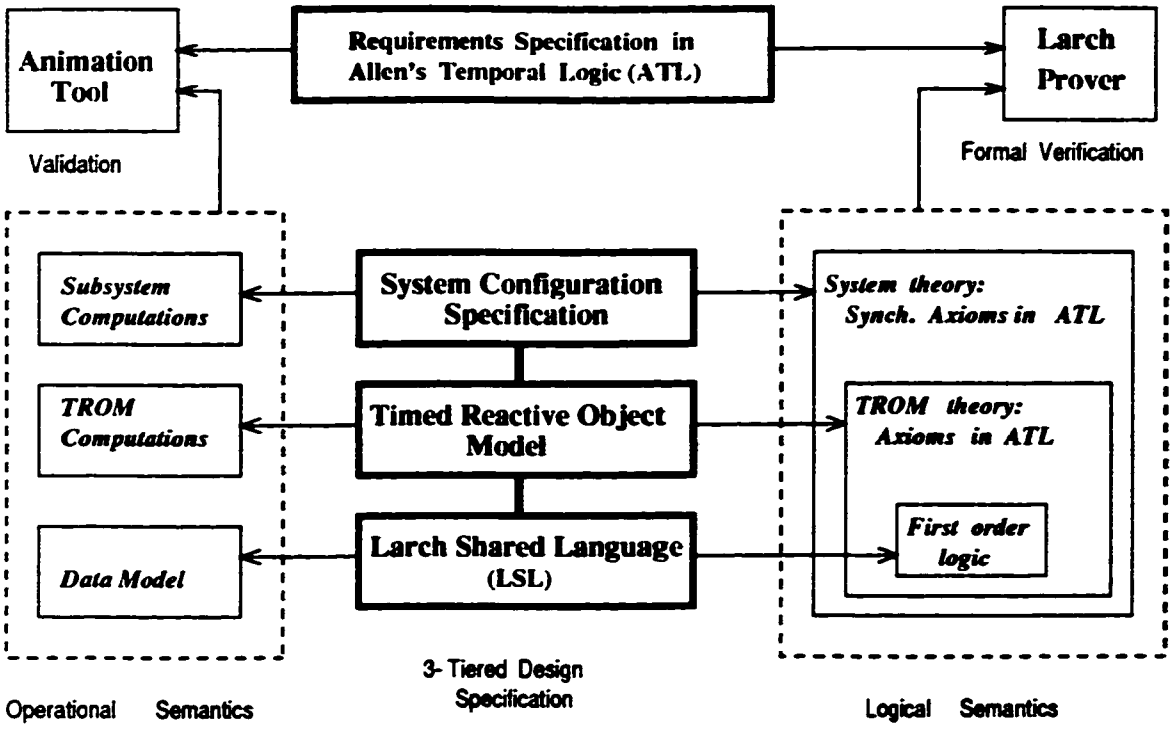


Figure 1: An overview of the methodology.

with the transition; and 3) *port-condition*, specifying the port at which an interaction happens.

2.2.1 Tier 1 – Data Abstraction

This level specifies the data abstraction used in the class definition of the middle tier by means of one of the languages of Larch, the *Larch Shared Language*(**LSL**)[10]. Larch languages are formal specification languages geared towards the specification of the observable effects of program modules, particularly modules which implement abstract data types. Larch provides a two-tiered approach to specification:

- In one tier, a Larch Interface language(LIL) is used to describe the semantics of a program module written in a particular programming language. LIL specifications provide the information needed to understand and use a module interface. LIL doesn't refer to a single specification language but to a family of specification languages. Each specification language in the LIL family is designed for a

specific programming language. The LIL for C++ is called Larch/C++.
LIL specifications are used to specify the abstract state transformations resulting from the invocation of the operations of a module. These specification are written in a predicative language using pre- and post-conditions.

- In the other tier, the Larch Shared Language(**LSL**) is used to specify state-independent, mathematical abstractions which can be referred to in LIL specifications. These underlying abstractions, called *Traits*, are written in the style of an equational algebraic specification.
LSL is programming language independent and is shared by all LILs.

The unit of encapsulation in **LSL** is the trait. Figure 2 shows an **LSL** trait which specifies the properties of a set. This example is similar to a conventional algebraic specification in the style of [10]. A trait contains a set of operator declarations, or signatures, which follows the keyword **introduces**. A set of equational axioms follows the keyword **assert**. A signature consists of the sorts and the domain and range of an operator. The equational axioms specifies the behavior and constraints on the defined operators.

There are a few notable differences between Larch traits and conventional algebraic specifications:

- The name of a trait (e.g. SetTrait) is distinct from the name of all sort and operator identifiers defined in the trait (e.g. Set).
- The names of sorts are not explicitly declared. They are implicitly declared by appearing in a signature.
- Larch makes use of the clauses **partitioned by** and **generated by** to increase the expressive power of traits.
- The semantics of = and == are exactly the same in **LSL**; only their syntactic precedence differs to ensure that expressions parse in an expected manner without having to use parentheses. The operator = binds more tightly than the operator ==.

$Set(E, C)$: **trait**
includes *Integer*
introduces
$\{\} :\rightarrow C$
$insert : E, C \rightarrow C$
$member : E, C \rightarrow Bool$
$delete : E, C \rightarrow C$
$size : C \rightarrow Int$
$isEmpty : C \rightarrow Bool$
asserts
C **generated by** $\{\}, insert$
C **partitioned by** $member$
$\forall\ s : C, e, e_1, e_2 : E$
$\neg(member(e, \{\}))$
$member(e_1, insert(e_2, s)) == e_1 = e_2 \vee member(e_1, s)$
$size(\{\}) == 0$
$size(insert(e, s)) ==$ **if** $member(e, s)$ **then** $size(s)$ **else** $size(s) + 1$
$delete(e_1, insert(e_2, s)) ==$ **if** $e_1 = e_2$ **then** s **else** $insert(e_2, delete(e_1, s))$
$isEmpty(\{\})$
$isEmpty(s) == size(s) = 0$
$\neg isEmpty(insert(e, s))$
implies
$\forall\ e, e_1, e_2 : E, s : C$
$insert(e, s) \neq \{\}$
$insert(e, insert(e, s)) == insert(e, s)$
$insert(e_1, insert(e_2, s)) ==$
$insert(e_2, insert(e_1, s))$
converts $delete, member, size, isEmpty$
exempting
$\forall\ e : E$
$delete(e, \{\})$

Figure 2: **LSL** Trait - Set

- Equations of the form *term* == *true* can be abbreviated to *term*; thus the first equation in Figure 2 is an abbreviation for $\neg (e \in \{\}) == true$ and the third equation is an abbreviation for $isEmpty(\{\}) == true$.

- The semantics of Larch traits is based on multisorted first order logic with equality rather than on an initial, final or loose algebra semantics used by other algebraic specification languages . Each trait denotes a theory[1] in multisorted first-order logic with equality. The theory contains each of the trait's equations, the conventional axioms of first order logic with equality, everything which follows from them, and nothing else. This means that the formulas in the theory follow only from the presence of assertions in the trait - never from their absence. The theory of a trait can also be strengthened by adding a **generated by** or a **partitioned by** clause.

- A trait definition need not correspond to an abstract data type (ADT) definition since an LSL trait can define any arbitrary theory of multisorted first-order equational logic. For example, a trait can be used to define the first order theory of mathematical abstractions such as equivalence relations which do not correspond to abstract data types.

- LSL traits can be augmented with checkable redundancies in order to verify whether intended consequences actually follow from the axioms of a trait. These checkable redundancies are specified in the form of assertions which are included in the **implies** clause of a trait and can be verified using Larch Prover.

In the trait of Figure 2, the **generated by** clause states that all values of the sort Set can be represented by terms composed solely of the two operator symbols {} and *insert*. In other words, saying that sort C is **generated by** a set of operators, Ops, asserts that each term of sort C is equal to a term whose outermost operators is in Ops. The operators in the set Ops are referred to as the *generators* of the sort C. A **generated by** clause strengthens the theory of a trait by adding an inductive rule of inference which can be used to prove properties which hold for all Set values.

[1] A *theory* is a set of logic formulas having no free variables.

For LSL traits which define an ADT, there is a sort referred to as the *distinguished sort*, sometimes also called the *principal sort* or *data sort*. For example, for the trait of Figure 2 the distinguished sort is Set, which is the sort corresponding to the set ADT.

The **partitioned by** clause provides additional equivalences between terms. Intutively, it states that two terms are equal if they cannot be distinguished by any of the functions listed in the clause. For the Set example, this property could be used to show that order of insertion in the set is commutative. That is, it could be shown that the terms $insert(e1, insert(e2, s))$ and $insert(e2, insert(e1, s))$ are equal for all values of e1, e2: Int and s: int.

The **exempting** clause documents the absence of right-hand sides of equations for delete(e, {}); this incompleteness is dealt with in the interface specification. The **converts** and **exempting** clauses together provide a way to state that this algebraic specification is sufficiently complete. Intutively, what the **converts** and **exempting** clauses are saying is the following: the specification of the operators *delete, isEmpty* is complete in the sense that any term involving these operators can be reduced to terms not involving these operators. The only exception to this rule is for terms which involve a subterm of the form $delete(e, \{\})$.

LSL also provides a way of putting traits together, one of which is through an **includes** clause. A trait that includes another trait is textually expanded to contain all operator declarations, **constraints** clauses, **generated by** clauses, and axioms of the included trait. The meaning of the including trait is the meaning of the textually expanded trait. In the Set example, the signature and meaning of the '+' operator comes from the **Integer** trait. Boolean operators (true, false, $\neg$ (not), $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$) as well as some heavily overloaded opeartors (if-then-else, =) are built into the language; that is, traits defining these operators are implicitly included in every trait.

We discuss Larch/C++ in Chapter 5, in the content of specifying Abstract Syntax Tree structures.

2.2.2 Tier 2 – TROM Methodology

In this level, reactive objects are modeled using **TROMs**. Communications between **TROMs** occur at the ports linking the **TROMs**. A port has an unique port type, which dictates the set of messages and the message sequences allowed at that port. A **TROM** can have ports of different types and several ports of one type. When an event E occurs at a port P at time T, an activity is initiated which may take a finite amount of time to complete. Due to the occurence of an event at the port, the **TROM** may undergo a state change, triggers or outputs several time-constraint events.

A formal definition of **TROM**, as given by [1], is the following:

Definition 2.2.2.1 A **TROM** is an 8-tuple ($\mathcal{P}$, $\mathcal{E}$, Θ, $\mathcal{X}$, $\mathcal{L}$, Φ, Λ, Υ) such that:

- *$\mathcal{P}$ is a finite set of port-types with a finite set of ports associated with each port-type. A distinguished port-type is the null-type P_0 whose only port is the null port o.*
- *$\mathcal{E}$ is a finite set of events and includes the silent-event* tick. *The set $\mathcal{E}$-tick is partitioned into three disjoint subsets: $\mathcal{E}_{in}$ is the set of input events, $\mathcal{E}_{out}$ is the set of output events, $\mathcal{E}_{int}$ is the set of internal events. Each $e \in (\mathcal{E}_{in} \cup \mathcal{E}_{out})$, is associated with a unique port-type $P \in \mathcal{P}$-P_0.*
- *Θ is a finite set of states. $\theta_0 \in \Theta$, is the initial state.*
- *$\mathcal{X}$ is a finite set of typed attributes. The attributes can be of one of the following two types: i) an abstract data type supporting a data model; ii) a port reference type.*
- *$\mathcal{L}$ is a finite set of LSL traits introducing the abstract data types used in $\mathcal{X}$.*
- *Φ is a function-vector (Φ_s, Φ_{at}) where,*
  - *Φ_s: $\Theta \rightarrow 2^\theta$ associates with each state θ a set of states, possibly empty, called substates. A state θ is called atomic, if $\Phi_s(\theta) = \varnothing$. By definition, the initial state θ_0 is atomic. For each non-atomic state θ, there exists a unique atomic state $\theta^* \in \Phi_s(\theta)$, called the entry-state.*

- Φ_{at}: $\Theta \rightarrow 2^{\mathcal{X}}$ *associates with each state* θ *a set of attributes, possibly empty, called active attribute set. At each state* θ*, the set* $\overline{\Phi_{at}}(\theta) = \mathcal{X} - \Phi_{at}(\theta)$ *is called the dormant attribute set of* θ*.*

- Λ *is a finite set of transition specifications including* λ_{init}*. A transition specification* $\lambda \in \Lambda - \lambda_{init}$*, is represented as* λ : $\langle\theta,\theta'\rangle$; $e(\varphi_{port})$; $\varphi_{en} \Longrightarrow \varphi_{post}$; *where:*

  - $\langle\theta,\theta'\rangle$*, where* $\theta, \theta' \in \Theta$ *are the source and destination states of the transition, respectively.*

  - $e(\varphi_{port})$ *where* $e \in \mathcal{E}$ *lables the transition;* φ_{post} *is an assertion on the attributes in* $\mathcal{X}$ *and a reserved variable* **pid**. **pid** *signifies the identifier of the port at which an interaction associated with the transition can occur. If* $e \in \mathcal{E}_{int} \cup$ **tick**, *then the assertion* φ_{port} *is absent and e is assumed to occur at the null-port o.*

  - $\varphi_{en} \Longrightarrow \varphi_{post}$*, where* φ_{en} *is the enabling condition and* φ_{post} *is the post-condition of the transition.* φ_{en} *is an assertion on the attributes in* $\mathcal{X}$*, primed attributes in* $\Phi_{at}(\theta')$ *and the variable* **pid** *specifying the data computation associated with the transition.*

  For each $\theta \in \Theta$*, the silent-transition* $\lambda_{so} \in \Lambda$ *is such that,*
  λ_{so}: $\langle\theta,\theta\rangle$; **tick**; *true* $\Longrightarrow \forall x \in \Phi_{at}(\theta)$: x=x';
  The initial-transition λ_{init} *is such that* λ_{init}: $\langle\theta\rangle$; Create(); φ_{init} *where* φ_{init} *is an assertion on active-attributes of* θ_o*.*

- Υ *is a finite set of time-constraints. A timing constraint* $v_i \in \Upsilon$ *is a tuple* (λ_i, e_i', [l,u], Θ_i) *where,*

  - $\lambda_i \neq \lambda_s$ *is a transition specification.*

  - $e_i' \in (\mathcal{E}_{int} \cup \mathcal{E}_{out})$ *is the constrainted event.*

  - [l,u] *defines the minimum and maximum response times.*

  - $\Theta_i \subseteq \Theta$ *is the set of states where in the timing constraint* v_i *will be ignored.*

Class *< classname > < [@port-types] >*

Events : *< event list >*

State : *< initial-state, list-of-states >*

Attributes : *list of < {att : att-type} >*

Traits : *list of < {traitname [port-type, att-type]} >*

Attribyte-function : *list of < {set of states -> set of atts} >*

Transition Spec : *< init-lable: <init-state>; Create(); >*
list of < trans-lable: <source, destination>;
event(assertion); assertion -> assertion; >

Time-Constraints :
list of < lable: (trans-lable, event,[min, max], set of state) >

end

Figure 3: The Well-formed **TROM** Class Definition.

The grammar given in Chapter 4 is based on this formal definition.

To apply this formalism in defining a **TROM** class. We give a template, shown in Figure 3. A user will write a **TROM** class specification based on this grammar.

2.2.3 Tier 3 – Subsystem Specification

This level is the topmost tier which constitutes subsystem configuration specifications(SCS), describing the system architecture by succinctly specifying the interaction relationship that can exist between the objects in a system. It also defines the inheritance and subtyping by aggregating instantiated objects to form subsystems and systems. Subystems are components in a system architecture. Each subsystem encapsulates the association, interaction, and concurrent evolution of a collection of **TROM**s. Hence a subsystem specification includes one or more SCS definitions.

The template for a subsystem configuration specification is shown below.

SCS: < name >

Include: list of < other_SCS_name>

Instantiate: list of < object_instantiation>

Configure: list of < object_port_aggregation>

This 3-tiered design specification forms the main component in the framework. Due to this approach, the design specification framework not only provides an architectural specification of a system, but also forms a means for formally specifying detailed design of the system component. One of the main advantages of this methodology is the conciseness in specification when object-oriented structuring principles such as instantiation, inheritance and subtyping are used as part of the design.
Through a case study, we illustrate the specification in the second and the third tiers. We also use the example to bring out the features of our tool.

2.3 Train-Gate-Controller Example

Consider a generalized version of a railroad crossing system introduced in [5]. We emphasize on its real-time reactive behavior and generalized version of the object-oriented design point of view. We will use this case study throughout the thesis.

2.3.1 An Informal Description

A railroad crossing system consists of a collection of *trains* and *gates* servicing the roads crossing the train tracks. The gate should remain closed whenever a train goes past the crossing. In order to control the gates there exists a collection of *controllers* such that one controller controls each gate. A controller closes its associaed gate when it gets a "nearing signal" from a train and opens the gate once all the trains crossing the gate have left. A controller does this by receiving signals from the trains and transmitting necessary control signals to its associated gate. Thus the problem is more general than the one studied before in the following sense: more than one train can cross a gate simultaneously, probably through multiple parallel tracks; a train can independently choose the gate it is going to cross, probably based on its direction and zone of travel. A safety requirement for the system is that whenever a train is inside a gate, the gate should remain closed.
When a train is approaching a gate, the train starts sending a *Near* message to the

controller associated with the gate. While leaving the gate. the train informs the controller by an *Exit* message. A typical time constraint on the train is that there is a minimum delay of at least 2 units of time before the train gets into the gate after it sends the message *Near*. Furthermore, after the message *Near*, there is a maximum delay of 5 units before which the train should exit the crossing.

A controller, upon receiving a *Near* message from a train, sends the signal *Lower* to its associated gate to lower the gate. Similarly, following the receipt of an *Exit* message from the last train to leave the gate, the controller sends a *Raise* message to the gate. There are two time constraints associated with the controller. The controller should respond by sending: i) a *Lower* message to the gate within 1 unit of time after receiving the *Near* message; ii) a *Raise* message to the gate within 1 unit of time after receiving the *Exit* message from the last train to leave the gate.

A gate responds to a *Lower* message and a *Raise* message by closing and opening the gate, respectively. There is a minimum delay constraint of 1 unit for closing the gate and a minimum of 1 unit and maximum delay constraint of 2 units, for opening the gate.

We discuss a formal specification of the system using our model in the following paragraphs.

2.3.2 A Formal Model

There are three types of interacting components: *Train, Gate* and *Controller*. The instantiation relationship in object-orientation helps to specify the system using three class specifications one for each of the above component types. The class interaction diagram for the system is shown in Figure 4.

The class specifications of the three components together with their state diagrams are shown in Figures 5 and 6.

Each class specification describes the behavior of a component individually. The RailRoad system is modeled as a SCS by instantiating the objects from the three classes and linking them.

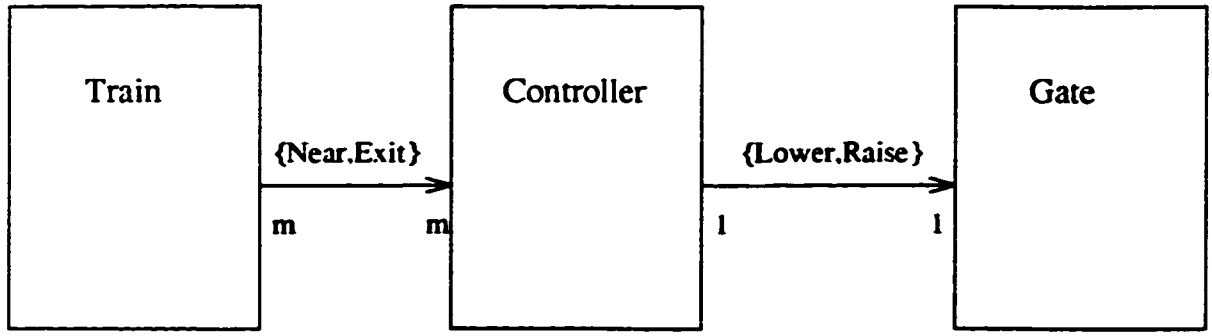
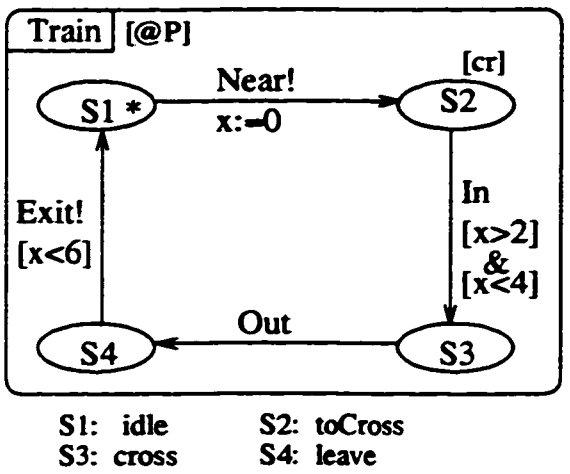


Figure 4: Class interaction diagram - RailRoad system

The train class supports a port-type, @P. A train starts in state S1. The port condition of the *Near* event in a train specifies that the train can non-deterministically select one of its ports of type @P for an interaction. A train class object models its intention to cross a specific gate(i.e., the gate associated with the controller which may be linked to the port selected). The attribute *cr* of states S2, S3 and S4 denotes that the further interaction of the train should be at the port it chose, until it exits the gate. The internal events *In* and *Out* signifies the start and end, respectively of the action of crossing a gate by the train. The two time constraints associated with a train are specified by the two tuples. The trigger event for both the time constraints is the event *Near*.

Initially the controller is in state C1. A controller supports ports of two types, @Q and @R. Implicitly, the ports of type @Q are used for interactions with train class objects and the ports of type @R are used for interactions with gate class objects. When in state C1, the controller responds to the input event *Near* at a port of type @Q by sending the event *Lower* at a port of type @R, within 1 unit of time. This corresponds to the signal from the first approaching train to enter the crossing after the gate was last opened. Subsequent *Near* events at other ports of type @Q mark the approaching signal from other trains, probably in parallel tracks. The attribute *inSet* associated with the states C2 and C3 denotes the set of ports at which an interaction involving *Near* has occured, and implicitly underscores those train class objects crossing the gate at that instant. It is obvious from the post conditions that, an insertion into the *inSet* and a deletion from it is done by the transitions *Near*

Class *train* [@*P*]
Events: *Near!P, Exit!P, In, Out*
State: **S1,S2,S3,S4*
Attributes: *cr* :@*P*
Attribute-function: $S1 -> \{\}$, $S3 -> \{\}$. $S3 -> \{\}$. $S1 -> cr$
Transition-Spec:
R_{init}: (*S1*); Create();
R1: (*S1,S2*); *Near*; true; true => $cr' =$ pid;
R2: (*S2,S3*); *In*; true => true;
R3: (*S3,S4*); *Out*; true => true;
R4: (*S4,S1*); *Exit*; $cr' =$ pid; true => true;
Time-constraints:
TC1: *R1,In*,[2,4],{};
TC2: *R1,Exit*,[0,6],{};
end



Class *Gate* [@*S*]
Events: *Lower?S, Raise?S, Down, Up*
State: **G1,G2,G3,G4*
Transition-Spec:
R_{init}: (*G1*); Create();
R1: (*G1,G2*); *Lower*; true; true => true;
R2: (*G2,G3*); *Down*; true => true;
R3: (*G3,G4*); *Raise*; true; true => true;
R4: (*G4,G1*); *Up*; true => true;
Time-constraints:
TC1: *R1,Down*,[0,1],{};
TC2: *R3,Up*,[1,2],{};
end

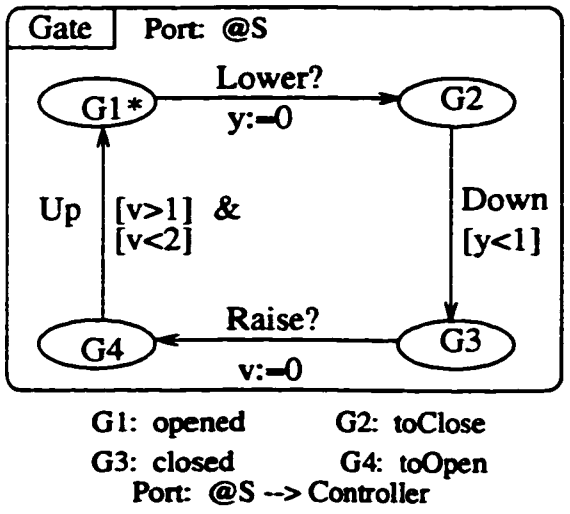
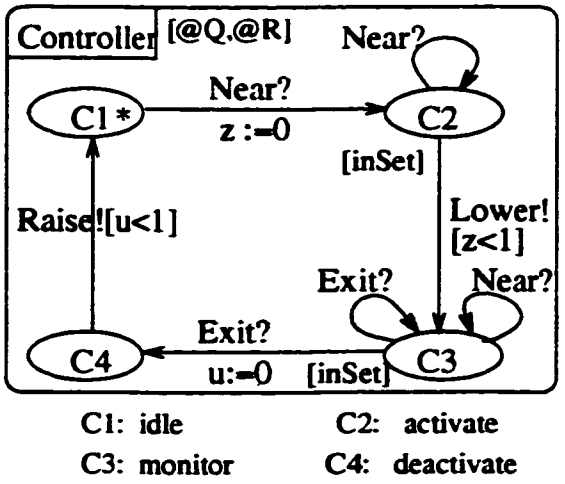


Figure 5: Class specifications for Train, Gate



Class *Controller* [@*Q*, @*R*]
Events: *Near?Q, Exit?Q, Lower!R, Raise!R*
State: **C1,C2,C3,C4*
Attributes: *inSet* : *TSet*
Traits: *Set*[@*Q*, *TSet*]
Attribute-function: *C1* − > {}; *C2* − > *inSet*; *C3* − > *inSet*; *C4* − > *inSet*
Transition-Spec:
R_{init}: (*C1*); **Create**();
R1: (*C1,C2*); *Near*; **true**; **true** => *inSet'* = *insert*(**pid**, *inSet*);
R2: (*C2,C2*), (*C3,C3*); *Near*; **NOT**(*member*(**pid**, *inSet*)); **true** => *inSet'* = *insert*(**pid**, *inSet*);
R3: (*C2,C3*); *Lower*; **true**; **true** => **true**;
R4: (*C3,C3*); *Exit*; *member*(**pid**, *inSet*); *size*(*inSet*) > 1 => *inSet'* = *delete*(**pid**, *inSet*);
R5: (*C3,C4*); *Exit*; *member*(**pid**, *inSet*); *size*(*inSet*) = 1 => *inSet'* = *delete*(**pid**, *inSet*);
R6: (*C4,C1*); *Raise*; **true**; **true** => **true**;
Time-constraints:
TC1: *R1,Lower*,[0,1],{};
TC2: *R5,Raise*,[0,1],{};
end

Figure 6: Class specifications for Controller

SCS *RailRoadSystem*
Instantiate:

t1 :: *Train*[@*P* : 2];
t2 :: *Train*[@*P* : 2];
t3 :: *Train*[@*P* : 2];
t4 :: *Train*[@*P* : 2];
c1 :: *Controller*[@*Q* : 4, @*R* : 1];
c2 :: *Controller*[@*Q* : 4, @*R* : 1];
g1 :: *Gate*[@*S* : 1];
g2 :: *Gate*[@*S* : 1];
Configure:

t1.@*P1* < − > *c1*.@*Q1*;
t2.@*P1* < − > *c1*.@*Q2*;
t3.@*P1* < − > *c1*.@*Q3*;
t4.@*P1* < − > *c1*.@*Q4*;
t1.@*P2* < − > *c2*.@*Q1*;
t2.@*P2* < − > *c2*.@*Q2*;
t3.@*P2* < − > *c2*.@*Q3*;
t4.@*P2* < − > *c2*.@*Q4*;
c1.@*R1* < − > *g1*.@*S1*;
c2.@*R1* < − > *g2*.@*S1*;
end

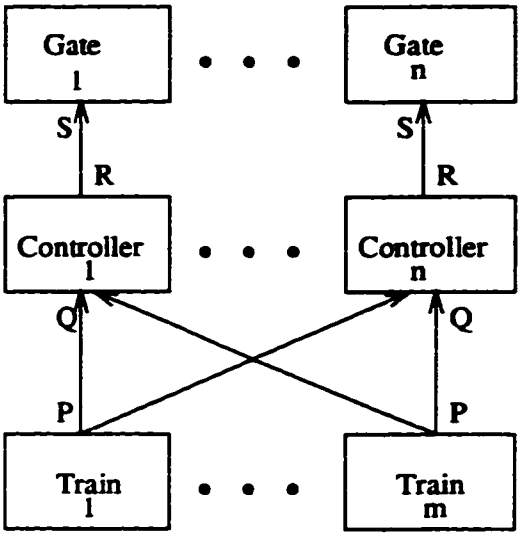


Figure 7: Subsystem specifications for RailRoadSystem

and *Exit*, respectively. Once the controller receives the event *Exit* at all the ports in the *inSet* (i.e., implicitly from all the trains which were crossing), the controller will send the event *Raise* at a port of type @R within 1 time unit as specified by the time-constraint. The port specification for the events *Near* and *Exit*, indicates that for each train instance any *Exit* event should be preceded by a *Near* event and every consecutive *Near* event should be interleaved by an *Exit* event.

The gate supports a port-type @S. A gate is open in start state G1 and closed in state G3. Interaction of the gate with its controllers is through its port involving events *Lower* and *Raise*. The events *Down* and *Up* are internal events and denote the end of the action, closing and opening, of the gate respectively. The reactions associated with the two time constraints of the gate are triggered by the events *Lower* and *Raise*, respectively.

The RailRoad subsystem is formed by composing objects instantiated from the above classes. A subsystem configuration specification is shown in Figure 7. The ability to specify objects independently and to configure a system/subsystem independently using the instantiated objects makes **TROM** suitable for specifying large systems.

Chapter 3

Architecture Design for Interpreter

To apply **TROM** methodology in practice, we need to develop a system enviornment – a tool, for user to write the **TROM** classes and Subsystem specifications and to be able to trace the interactive object behavior at run time. This tool consists of three major parts: User Interface, Interpreter and Simulator. The overall structure of the tool is shown in Figure 8.

This thesis focuses on developing an Interpreter which takes user's **TROM** class and Subsystem specifications and generates an internal representation (i.e., AST - abstract syntax tree) to be used by the Simulation Tool. It also focuses on designing an Axiom Generator which is used by the verification system and it is discussed in Chapter 7.

The architecture design for the Interpreter is shown in Figure 9.

The Interpreter will take the user input file and generate an internal data representation which is used during simulation. It also generates axioms which are used to verify system properties. The components of the Interpreter are the following:

- *Scanner* - used to identify input file and to generate tokens. It also does basic lexical analysis using **Flex**.
- *Parser* - clarifies tokens from scanner in order to check syntactic correctness by using **Bison**.

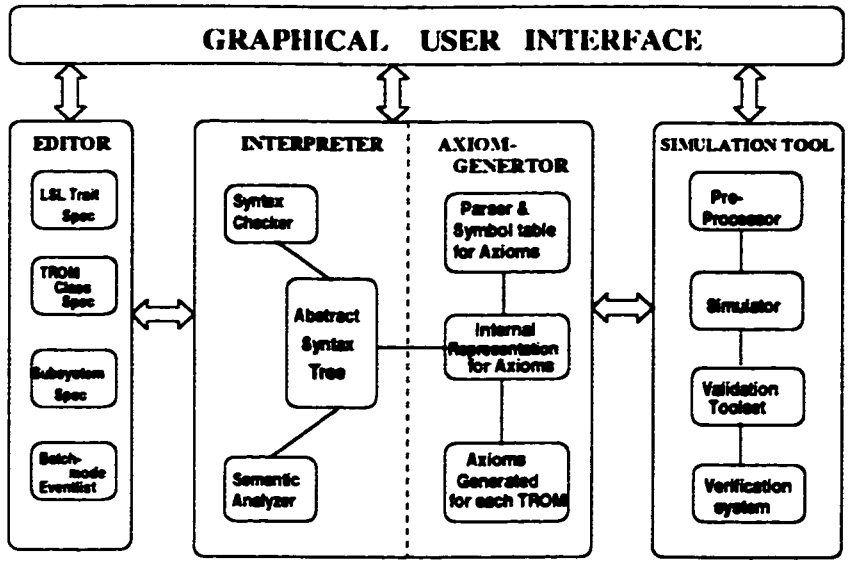


Figure 8: The Architecture of Overall System

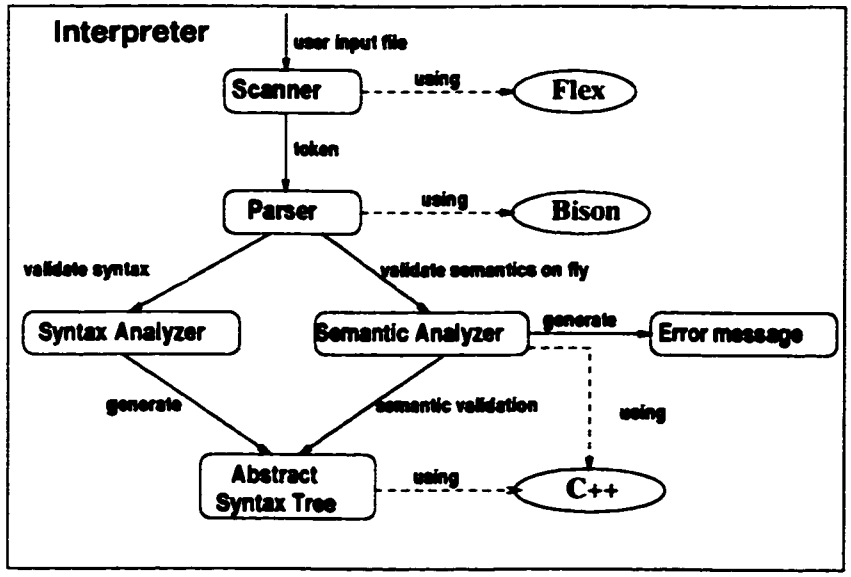


Figure 9: The Architecture for Abstract Syntax Tree

- *Syntax Analyzer* - uses a pre-defined grammar for **TROM** and SCS in order to validate syntactically correct tokens. Any error will terminate the program and is directly reported to the user by **Bison**.

- *Abstract Syntax Tree* - a data structure to represent the useful information from user input file(the detailed structure of the AST will be discussed in Chapter 5).

- *Semantic Analyzer* - a program written in C++ and used to do semantic checking. There are two types of semantic checking.

  i) *On the fly validation* - does simple semantic check at the same time with syntax checking. For example, duplicate names can be checked during on fly validation.

  ii) *AST validation* - does complex semantic checking after building up an AST. For example, type checking, linking to **LSL** traits,.., must be done at this stage.

- *Error Message* - generated by semantic analyzer in order to help user correct the input file. Every detected error will be kept in a file until the end of the analysis.

Chapter 4

Syntax Analysis for TROM class and Subsystems

The templates for **TROM** and Subsystem are given in Chapter 2. In this chapter, we focus only on the syntax and discuss the grammar for well-defined **TROM** classes and Subsystems.

4.1 A Formal Grammar for TROM class

The **TROM** class is composed of several members, shown in Table 1.

We will give a grammar for each of the **TROM** class member. Words shown in "sans serif type" font are keywords and case sensitive. CHAR(10) in the grammar means that at most 10 characters are allowed.

The specification of *Train*-**TROM**, *Gate*-**TROM** and *Controller*-**TROM** as given in Chapter 1 are consistant with the grammar given bellow. From Definition 2.2.2.1 and class specifications shown in Figure 3, we abstract a formal grammar for **TROM**:

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

Table 1: **TROM** class Description

class	::=	**Class** <*class_name*> [<*port_types*>] NL
port_types	::=	@<*port_type_name*> | @<*port_type_name*>, <*port_types*>
class_name	::=	CHAR(10)
port_type_name	::=	CHAR(10)

Table 2: A grammar for **TROM** class member – Class

4.1.1 TROM class member – *Class*

Class is a header which describes the name and the port types of a **TROM**. The format is:

- The keyword **Class** followed by class name and *Port types*.
- *Port types* are preceded by the symbol @, within square brackets, and separated by commas.

The grammar for *class* is shown in Table 2.

4.1.2 TROM class member – *Event*

Event enumerates the set of events associated with the class. The format is:

- The keyword **Events** followed by a colon and list of events.
- Events are separated by commas and must belong to one of the three types:
  - Input event: an event name followed by the symbol ? and a *port type* for the event.
  - Output event: an event name followed by the symbol ! and a *port type* for the event.
  - Internal event is not marked.

The grammar for *Event* is shown in Table 3.

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	::=	CHAR(10)
port_type_name	::=	CHAR(10)

Table 3: A grammar for **TROM** class member - 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	::=	CHAR(10)

Table 4: A grammar for **TROM** class member - States

4.1.3 TROM class member – *State*

State enumerates the abstract states of the class. The format is:

- The keyword **State** followed by a colon and a list of states.
- States are separated by commas and contains three special types:
  - *Initial state*: state name preceded by the symbol *.
  - *Substates*: list of state lables separated by commas within parentheses. The states should follow their parent state.
  - *Initial substate*: substate name preceded by the symbol *.

The grammar for *State* is shown in Table 4.

4.1.4 TROM class member – *Attribute*

Attribute describes the set of attributes belonging to the class together with their

attributes	::=	**Attributes:** <*att_list*> NL | ϵ
att_list	::=	<*attribute*> | <*attribute*>: <*att_list*>
attribute	::=	<*att_name*> : @<*port_type_name*> | <*att_name*> : <*trait_type_name*>
att_name	::=	CHAR(10)
trait_type_name	::=	CHAR(10)
port_type_name	::=	CHAR(10)

Table 5: A grammar for **TROM** class member – Attributes

types. The format is:

- The keyword **Attributes** followed by a colon and a list of attributes.
- Attributes are separated by semi-colon.
- An attribute is represented by its name, followed by a colon and a type of the attribute.
- The type of an attribute is either a *port type* or an abstract data type associated with a **LSL** trait.
- The *Attribute* description is optional.

The grammar for *Attribute* is shown in Table 5.

4.1.5 TROM class member – *Trait*

Trait imports the behavior of a specified data model which belongs to the corresponding trait defined in the **LSL** tier. Any trait reference type specified in *Attribute* section must match one of the parameters listed in a trait in this section. This defines the link between two tiers i.e., the data abstraction and the class definition. The format is:

- The keyword **Traits** followed by colon and list of traits together with their parameters.

traits	::=	**Traits:** <*trait_list*> NL | ε
trait_list	::=	<*trait*> | <*trait* > ; <*trait_list* >
trait	::=	<*trait_name*> [<*arg_list*>, <*trait_type_name*>]
arg_list	::=	<*arg*> | <*arg*>,<*arg_list*>
arg	::=	<*trait_type_name*> | @<*port_type_name* >
trait_name	::=	CHAR(10)
trait_type_name	::=	CHAR(10)
port_type_name	::=	CHAR(10)

Table 6: A grammar for **TROM** class member –Traits

- Traits are separated by commas.
- A trait is represented by its name followed by its parameters.
- The parameters for a trait are enumerated within square brackets and are separated by commas.
- The *Trait* section is optional.

The grammar for *Trait* is shown in the Table 6.

4.1.6 TROM class member – *Attribute-function*

Attribute-function defines the association of attributes to states, thus partitioning the domain of attributes into active and dormant parts at each state. The format is:

- The keyword **Attribute-function** followed by colon and list of attribute-functions.
- Attribute-functions are separated by semi-colons.
- An attribute-function is presented as follows:
  - A state name followed by the symbol – >
  - One or more names of attributes separated by commas and enumerated within brackets (the symbol {} representing an empty set).

att_funcs	::=	**Attribute_function:** <*att_func_list*> | ε
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*> | ε
att_name	::=	CHAR(10)
state_name	::=	CHAR(10)

Table 7: A grammar for **TROM** class member – Attribute-functions

- The *Attribute-function* section is optional.

The grammar for *Attribute-functions* is shown in Table 7.

4.1.7 TROM class member – *Transition-Spec*

Transition-Spec describes the transition specification for each transition in the state machine. The format is:

- The keyword **Transition Spec** is followed by a colon and the list of transitions which start with initial transition followed by other transitions.
- Each transition starts at a new line with its name and finishes with a semi-colon.
- An initial transition is specified as follows:
  - The initial transition name followed by a colon
  - The symbol < followed by a name of starting state, the symbol > followed by a semi-colon
  - **Create()**;
  - One or more assertions separated by **AND** or **OR** within parentheses. An assertion which is an expression in one of the following format:
    - ⋆ The constant **true**.

* An attribute name defined in the *Attributes* section plus a superscript' which means the post condition of the attribute followed by the operater = and either a function as defined in the **LSL** trait with the corresponding arguments enclosed within parentheses, the reserved **pid** or the same attribute name without superscript' .

- A semi-colon.
- The initial transition specification is optional

- Other transitions have the following format:
  - The transition name followed by a colon
  - List of state pairs are separated by commas. Each state pair is specified as (S1, S2), where S1, S2 are state names.
  - A semi-colon followed by an event name
  - A semi-colon followed by a port condition
  - The port condition syntax is: One or more assertions separated by **AND** or **OR** within parantheses. An assertion is a *Boolean* expression which is in one of the following formats:
    * The constant **true**.
    * A *Boolean* function as defined in **LSL** traits with the corresponding arguments enclosed within parenthesis. An argument can be an attribute defined in *Attributes* section or the reserved **pid**. The *Boolean* function can be preceded by the keyword **NOT**.
    * An Integer expression followed by a *Boolean* operator and another Integer expression. An Integer expression can be an *Integer* or a function returning an *Integer* as defined in the **LSL** trait with the corresponding arguments enclosed within parentheses. *Boolean* operators are =, ! =, <, >, <=, >=.
  - A semi-colon followed by an enabling condition which has the same syntax as port condition described above.

- The symbol=> followed by a post condition.
- The post condition syntax is: One or more assertions separated by **AND** or **OR** within paranthcses. An assertion is a *Boolean* expression which is in one of the following format:
  - ⋆ The constant **true**.
  - ⋆ An attribute name defined in the *Attributes* section plus a superscript′ which means the post condition of the attribute followed by the operater = and either a function as defined in the **LSL** trait with the corresponding arguments enclosed within parentheses, the reserved **pid** or an attribute name without superscript′.
- A semi-colon.

The grammar for *Transition Specification* is shown in Table 8.

4.1.8 TROM class member – *Time-Constraint*

Time-constraints describe the timing constraints associated with the transitions. The timing constraints ensure that the class will not keep the resource unallocated indefinitely. The syntax is according to the following format.

- The keyword **Time-Constraints** followed by a colon and a list of Time-constraints which are separated by semi-colons.
- Each specific Time-constraint starts at a new line, and has the following format:
  - A Time-constraint name followed by a colon
  - A transition specification name followed by a comma
  - An event name followed by a comma
  - A left square bracket
  - An integer for the minimum response time followed by a comma
  - An integer for the maximum response time

tran_specs	::=	**Transition Spec:** NL <*tran_spec_init*> NL <*tran_spec_list*> NL
tran_spec_init	::=	<*tran_spec_name*>: <*state_name*>; **Create();** <*assertion_op*>; | ϵ
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*>) | <*event_name*>
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'	::=	CHAR(10)
att_name	::=	CHAR(10)
state_name	::=	CHAR(10)
event_name	::=	CHAR(10)
LSL_func_name	::=	CHAR(10)

Table 8: A grammar for **TROM** class member – Transition Specifications

Time_Constriants	::=	**Time_Constraints:** NL <*constriants*>
constriants	::=	<*constriant*> NL | <*constriant*> NL <*constriants*>
constriant	::=	<*time_cons_name*>: (<*tran_spec_name*>, <*event_name*>, [<*min*>, <*max*>], {<*states*>})
states	::=	<*state_name*> | <*state_name*>, <*states*> | ϵ
state_name	::=	CHAR(10)
time_cons_name	::=	CHAR(10)
tran_spec_name	::=	CHAR(10)
event_name	::=	CHAR(10)
min	::=	NAT
max	::=	NAT

Table 9: A grammar for **TROM** class member – Time-Constraints

SCS ::= <*scs*> <*include*> <*instantiate*> <*configure*> end

Table 10: SCS Description

- A right square bracket followed by a comma
- Ignoring-states: list of states separated by commas and enumerated within brackets. The symbol {} representing an empty set.

The grammar for *Time-Constraint* is defined in the Table 9.

4.2 A Formal Grammar for Subsystem

A Subsystem Configuration Specification(SCS) is used to specify a system/subsystem by aggregating instantiated objects. It can also be used to build large systems by composing subsystems. The syntax for a SCS is given in Table 10.

As shown in Table 10, a SCS is composed of several members. We will give a grammar for each of the SCS member.

The subsystem configuration specification for *Train*, *Gate* and *Contorller* as given in chapter 1 is consistant with the grammar given bellow.

scs	::=	SCS <*scs_name*> NL
scs_name	::=	CHAR(10)

Table 11: A grammar for SCS member - SCS-name

include	::=	Include: <*scs_name_list*> NL | ϵ
scs_name_list	::=	<*scs_name*>; | <*scs_name_list*>
scs_name	::=	CHAR(10)

Table 12: A grammar for SCS member - Include

4.2.1 Subsystem member – *SCS-name*

SCS-name is a header which describes the name of the SCS. The format is:

- The keyword SCS followed by scs name

The grammar for *SCS-name* is shown in Table 11.

4.2.2 Subsystem member – *Include*

Include is used for importing subsystem definitions from other subsystem configuration specification. This provides modularity and makes it easier to reuse pieces of specification. The format is:

- The keyword Include followed by a colon and a list of other SCS names which are separated by commas.
- The *Include* is optional.

The grammar for *Include* is shown in Table 12.

4.2.3 Subsystem member – Instantiate

Instantiate is used to define instantiation relationship between objects and their classes. The format is:

instantiates	::=	**Instantiate:** <*inst_list*> NL | ϵ
inst_list	::=	<*instantiate*> | <*instantiate*>: <*inst_list*>
instantiate	::=	<*obj_name*> :: <*trom_name*> [<*port_card_list*>]
port_card_list	::=	<*port_card*> | <*port_card*>, <*port_card_list*>
port_card	::=	<*port_type_name*> : <*cardinality*>
obj_name	::=	CHAR(10)
port_type_name	::=	CHAR(10)
trom_name	::=	CHAR(10)
cardinality	::=	NAT

Table 13: A grammar for SCS member – Instantiate

- The keyword **Instantiate** followed by a colon and list of instantiated objects which are separated by semi-colon.
- object name followed by the symbol ::
- A **TROM** class name followed by a left square bracket.
- A port type, followed by a colon, and the cardinality of ports for that port type. If there are more than one port type, a comma separates the tuples of port type and cardinality.
- A right square bracket, followed by a point.
- The *Instantiate* is optional.

The grammar for *Instantiate* is shown in Table 13.

4.2.4 Subsystem member – *Configure*

Configure is used to define a system/subsystem using objects specified in the *Instantiate* clause and subsystem specifications imported through the *Include* clause. It also links the ports of various interacting objects/subsystems using compositions.The format is:

- The keyword **Configure** followed by a colon and list of object-port links which are separated by semi-colon. Object-port link has the following syntax:

configure	::=	**Configure:** <*obj_port_list*>
obj_port_list	::=	<*obj_pork_link*> ; | <*obj_pork_link*>; <*obj_port_list*>;
obj_pork_link	::=	<*obj_name*>. @<*port_name*> < − > <*obj_name*>. @<*port_name*>
obj_name	::=	CHAR(10)
port_name	::=	CHAR(10)

Table 14: A grammar for SCS member – Configure

- A pair of an instantiated object name followed by a symbol . and a symbol @ followed by a port name associated with the object.
- The symbol < − >
- Another pair of an instantiated object name followed by a symbol . and a symbol @ followed by a port name associated with the object.

The grammar for *Configure* is shown in Table 14.

4.3 A Formal Grammar for *Simulation Event*

sel	::=	**SEL:** <*s_event_list*>
s_event_list	::=	<*s_event*> | <*s_event*>; <*s_event_list*>
s_event	::=	<*event_name*>, <*obj_name*>, <*port_name*> <*occur_time*>
event_name	::=	CHAR(10)
obj_name	::=	CHAR(10)
port_name	::=	CHAR(10) | **NULL**
occur_time	::=	NAT

Table 15: A grammar for Simulation Event List

A *Simulation Event* is used to stimulate the user defined system and to trace the system behaviour. The Simulation Events are accepted as tuples, each appearing on a line. The format is:

- The keyword **SEL** followed by a colon and a list of simulation events which are separated by semi-colons. Each simulation event has the following syntax:

- An event name, followed by a comma.
- A **TROM** class object name, followed by a comma.
- A port name, followed by a comma. If the event is an internal event, the null port is specified by the reserved word NULL.
- A natural number specifying the time at which the event occurs, relative to the start time.

The grammar for Simulation Event List is shown in Table 15.

Chapter 5

Internal Representation – Abstract Syntax Tree

The Abstract Syntax Tree is an object-based data structure which contains objects that can be accessed during simulation. The animation system has four kinds of input: **LSL** traits, **TROM**, SCS, and Simulation Event List. Therefore, four kinds of Abstract Syntax Trees are constructed. The data structure of each AST might be different, depending on its usage. This section will focus on the detailed data structure design for each AST and also discuss the formal software specification using Larch/C++ specification language.

5.1 The TROM Abstract Syntax Tree Data Structure

As we mentioned in the previous chapter, a **TROM** input may contain one or several **TROM** classes. We should construct an AST for each **TROM** class.

The **TROM** AST data structure is shown in Figure 10.

As shown in Figure 10, the **TROM** AST is a data construct which is a collection of **TROM** class member objects. There is a container for each type of class member object which can be implemented by a linked list. The **TROM** AST is unique and it can be identified by its class-name attribute. Some of the **TROM** class members

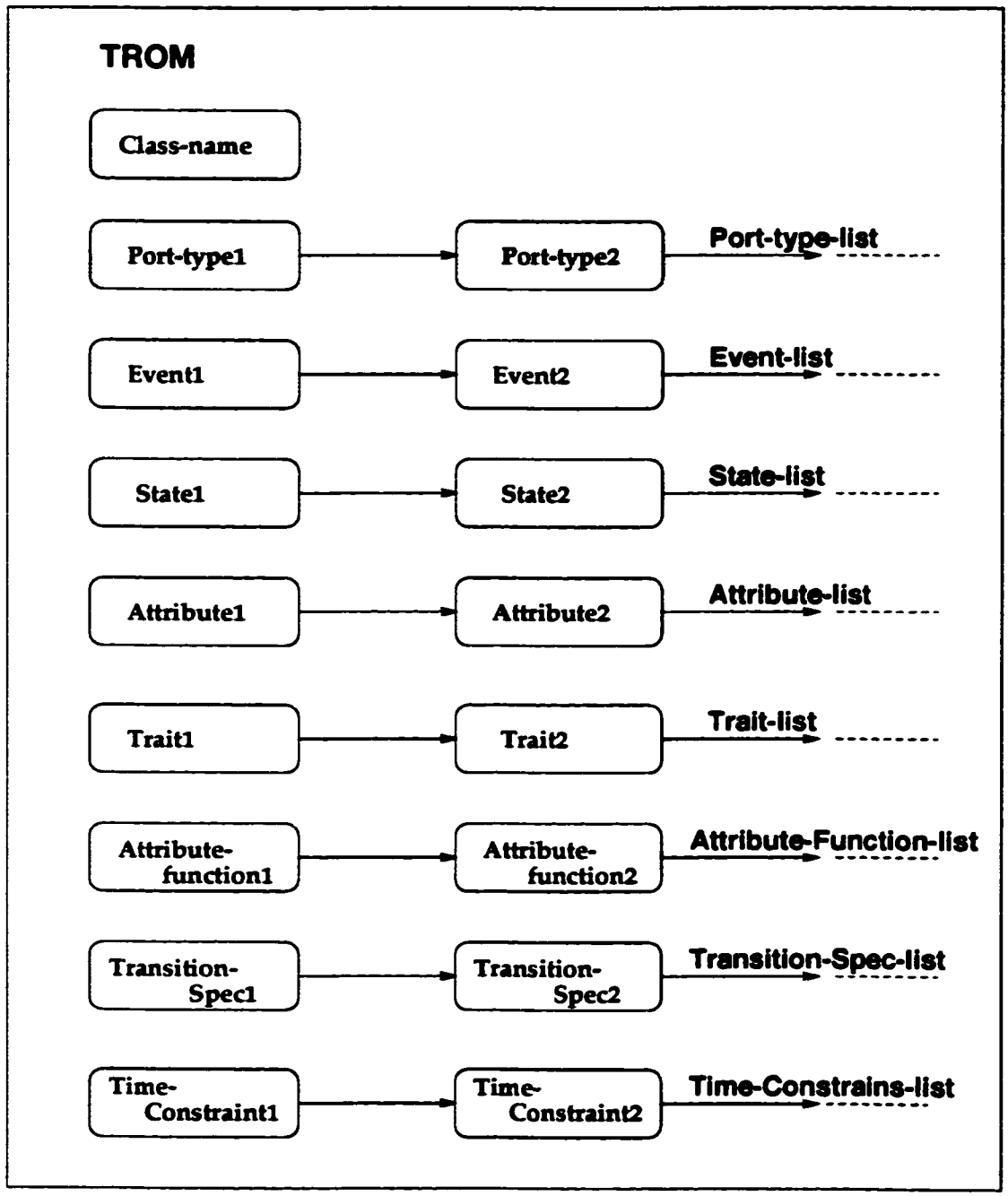


Figure 10: **TROM** Abstract Syntax Tree Structure.

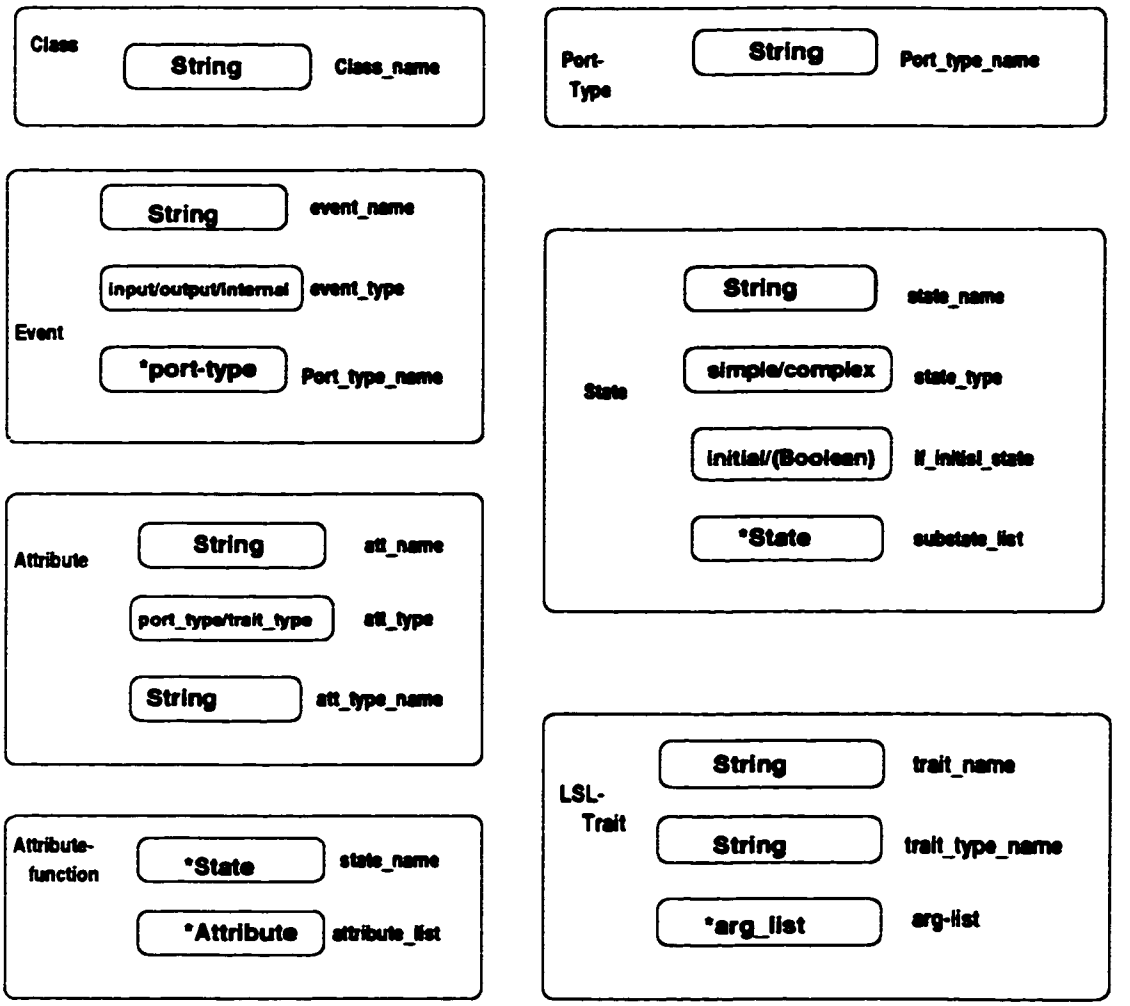


Figure 11: Data Structure of **TROM** class members.

such as class-name, port-type list, event list are mandatory. So, their corresponding containers must contain at least one element. Others such as traits, attribute-function are optional. So, their containers can be empty.

The data structure of **TROM** class members are described in detail in the following subsections (see Figure 11, Figure 12 and Figure 13).

5.1.1 General Description for TROM AST class member containers

Except the Class_name. all the **TROM** AST class members are represented by a container data type, which we said can be implemented by a linked-list. We named

them following the convention: X_list is a container of elements of type X. It can be implemented by a list (linked-list) of X-type data.

For example, **Event_list** is a container of elements of type Event. To apply the Object-Oriented system design principles, the container is an abstract data type which encapsulates the implementation details. We design the container class which supports the following standard access functions:

- *create*: Create an empty data container
- *isEmpty*: Return a boolean value to assert if the container is empty
- *remove*: Remove a specific element from the container
- *insert*: Insert an element into the container at a specific location
- *append*: Add an element into the container at the end
- *find*: Find out a specific element from the container
- *getFirst*: Retrieve the first element from the container
- *getNext*: Retrieve an element from the container next to the current location
- *destroy*: Destroy the container and all its elements

The set of container management and navigational functions allow the fellow researchers to use the **TROM** AST without knowing the implementation details of their data structures.

5.1.2 Class name

The *class name* is represented by a string variable which contains an unique **TROM** class identifier. The *class* name is mandatory in **TROM** class AST.

5.1.3 Port-type-name

The *port-type-name* is represented by a string variable which is not duplicable in the **port-type_list**. The **port-type_list** is mandatory in **TROM** class AST.

5.1.4 Event

The *event* is represented by a data structure in the **event_list**. The data structure contains the following data members:

- *event_name*: a string variable which is not duplicable in the event_list.
- *event_type*: an enumerated variable with the values input, output, internal.
- *port_type_name*: a reference to the existing *port_type_name* in the **port-type_list** or **null** if the *event_type* is internal.

The **Event_list** is mandatory in **TROM** class AST.

5.1.5 State

The *State* is represented by a data structure in the **State_list**. The data structure contains the following data members:

- *state_name*: a string variable which is not duplicable in the State_list.
- *state_type*: an enumerated variable with the values simple, complex. The complex state type means the state has a set of substates.
- *if_initial_state*: a boolean variable to indicate if it is the initial state.
- *Substate_list*: a container of substates of the **State_list** type. It must be empty for simple state type and it must contain at least two different states for complex state types.

The **State_list** is mandatory in **TROM** class AST.

5.1.6 Attribute

The *Attribute* is represented by a data structure in the **Attribute_list**. The data structure contains the following data members:

- *att_name*: represents an attribute name and is a string variable, which is not duplicable in the **Attribute_list**.

- *att_type*: represents an attribute type and is an enumerated variable with the values { *port_type*}, or { *trait_type*}. The *trait_type* is defined in the **Trait_list** and *port_type* is defined in **Port-type_list**

- *att_type_name*: represents an attribute type name and is a reference to an existing **Port-type-list** or **Trait-list**.

The **Attribute_list** is mandatory in **TROM** class AST.

5.1.7 Trait

The *Trait* is represented by a data structure in the **Trait_list**. The data structure contains the following data members:

- *trait_name*: represents a trait name and is a string variable, which is not duplicable in the **Trait_list**.

- *trait_type_name*: represents a trait type name and is a string variable, which is not duplicable in the **Trait_list**.

- *arg_list*: a container of arguments set associated with the trait. It can be a reference to an existing *port_type_name* or *trait_type_name* and at least contains one argument.

The **Trait_list** is optional in **TROM** class AST.

5.1.8 Attribute-function

The *Attribute-function* is represented by a data structure in the **Attribute-function_list**. The data structure contains the following information:

- *state_name*: a reference to an existing state in **State_list**.

- *attribute_name_list*: a reference container to the existing **Attribute_list** associated with the state. *attribute_name_list* associated with a state can be empty.

The **Attribute-function_list** is optional in **TROM** class AST.

5.1.9 Assertion Tree Data Structure

We have seen that in the grammar for Transition Specification, the port_conditions, enabling_conditions and post_conditions are all instances of a generic type, an assertion expression. The assertion expression can be represented by and stored in a specific container type – a **binary tree**. It can be evaluated by traversing the binary tree in inorder.

We can name the various Transition_Spec assertions by the following convention:

X_tree is a binary tree container of elementary assertions for representing the conditions of type X.

For example, *port_conditions_tree* is a container of elementary assertions for representing the assertion of the type *port_condition.*

An assertion expression represented by a binary tree is illustrated in Figure 12.

As shown in Figure 12, each node in the assertion expression tree is a data type of either an operator or an operand. The root node of the tree or any of its branches are always of type operator. Only the leaves of the assertion expression tree can be and must be of type operand. If the node is not a leaf, it may have a left or a right child subtree. The leaf node has nil values for its left and right child.

In the Object-Oriented system design, the binary tree container is an abstract data type which encapsulates the implementation details. We design the binary tree container class which supports the following standard access functions:

- *create*: Create an empty binary tree data container
- *isEmpty*: Return a boolean value to assert if the binary tree container is empty
- *addLeft*: Add an element into the binary tree container as the left child of the current element
- *addRight*: Add an element into the binary tree container as the right child of the current element

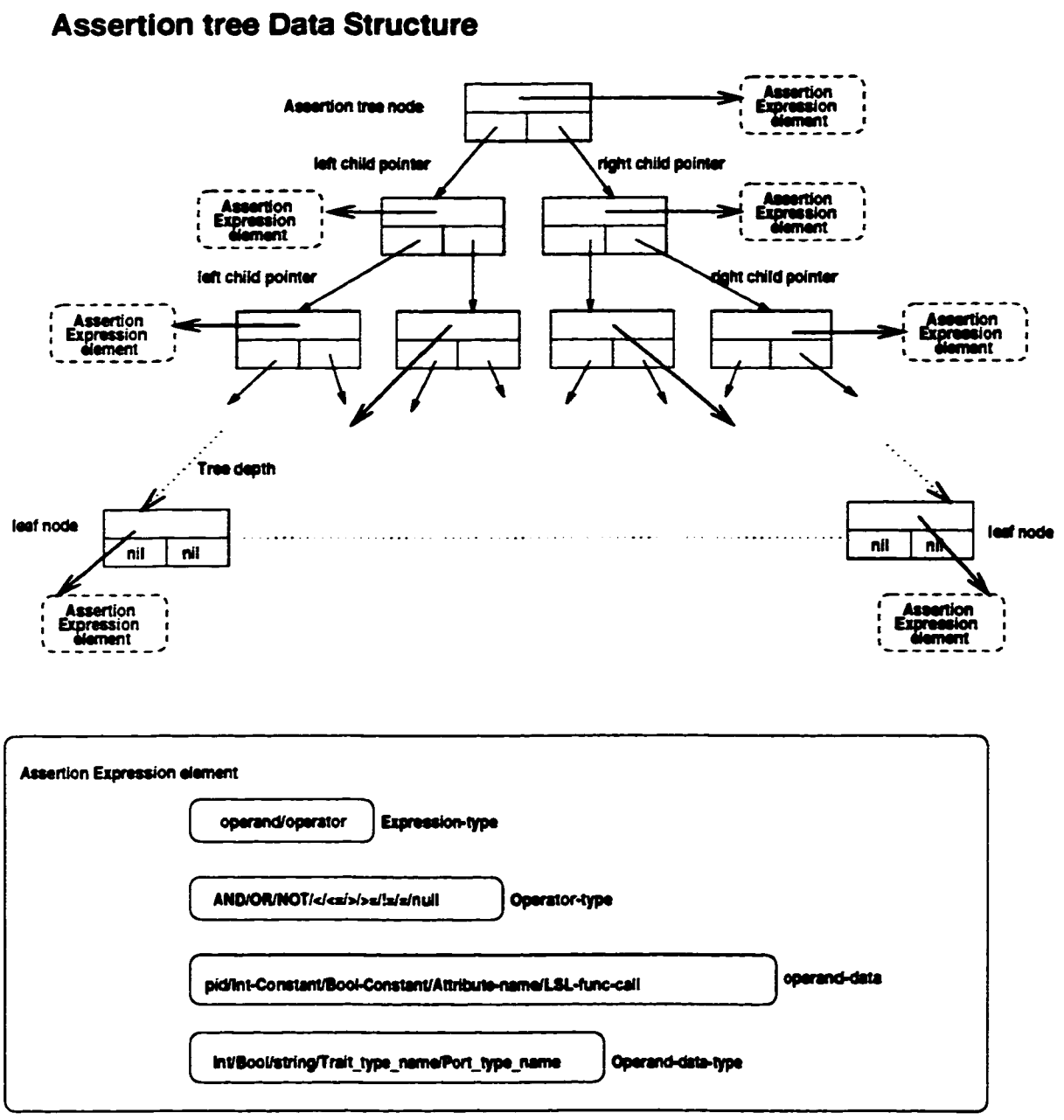


Figure 12: Data Structure of an Assertion tree.

- *remove*: Remove a specific subtree rooted by the current element from the binary tree container
- *insert*: Insert an element into the binary tree container before the leftmost element
- *append*: Add an element into the binary tree container after the rightmost element
- *find*: Find out a specific element from the binary tree container
- *getFirst*: Retrieve the first element from the binary tree container following the leftmost deepest first traversing rule
- *getNext*: Retrieve an element from the binary tree container next to the current location following the leftmost deepest first traversing rule
- *destroy*: Destroy the binary tree container and all its elements

The set of container management and navigational functions allow the fellow researchers to use the **TROM** AST without knowing the implementation details of their data structures.

5.1.10 Transition-Spec

The *Transition-Spec* is represented by a record data structure in the **Transition-Spec_list**. The data structure contains the following data members:

- *transition_lable*: a string variable which is not duplicable in the **transition-Spec_list**.
- *if_initial_tran*: a boolean variable to indicate if it is the initial transition specification.
- *source_state*: represents a starting state of this transition and is a reference to the existing state in the **State_list**.

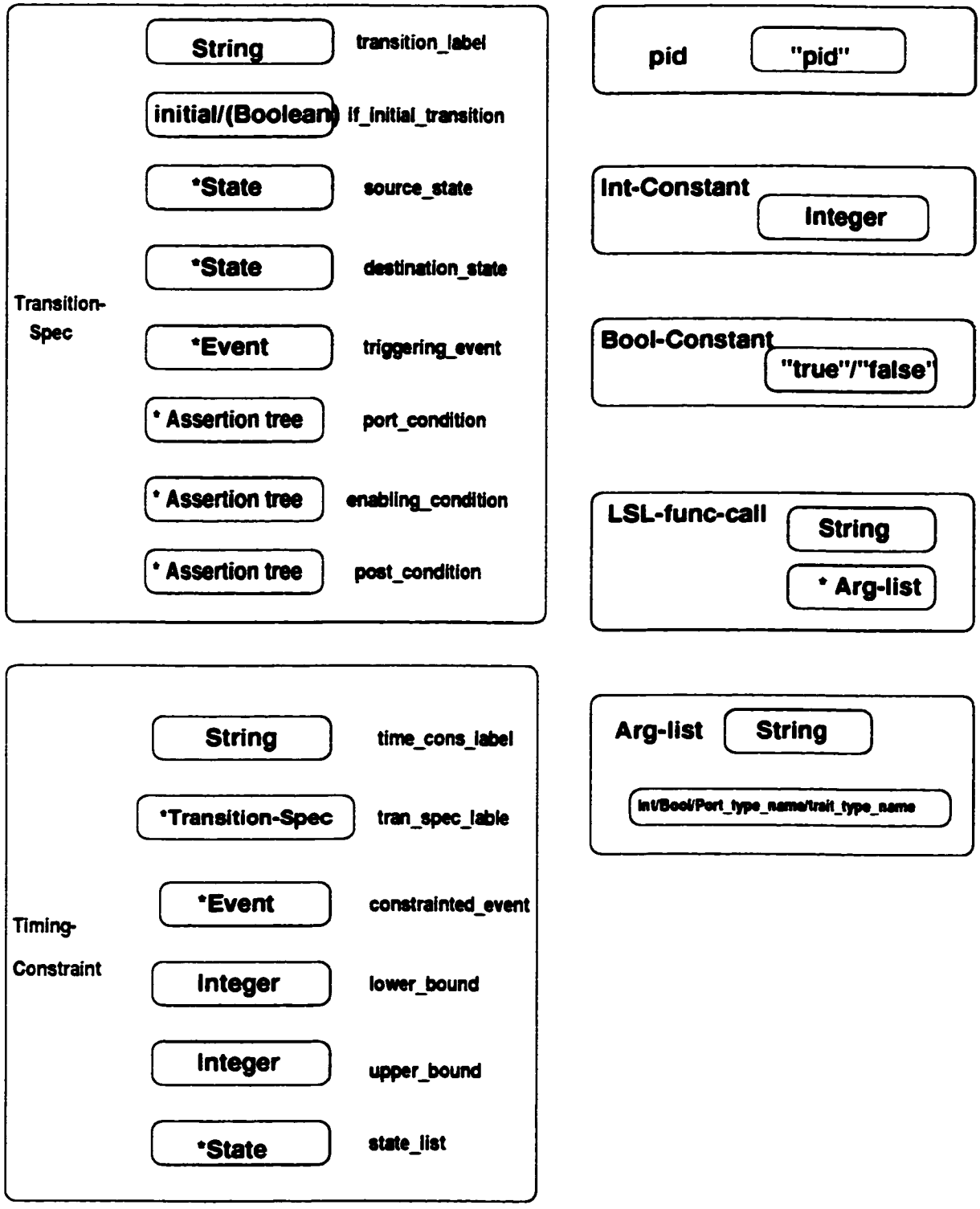


Figure 13: Data Structure of **TROM** class members.

- *destination_state*: represents a destination state of this transition and is a reference to the existing state in the **State_list**.

- *event*: a reference to the existing event in the **Event_list**.

- *port_condition*: is an assertion tree corresponding to the event. If the event is an internal event, port_condition is always set to **True**. The assertion tree is a binary tree and each *Assertion Expression element* has the following data members:

  - *Expression-type*: a boolean variable to denote if the element is an operand or an operator.

  - *Operator-type*: an enumerated variable denoting the operator in this node. It is null if the *Expression-type* is operand.

  - *Operand-data*: an enumerated variable with the values of **pid**, **integer**, **boolean**, **attribute-name**, **LSL-func-call**. It is null if the *Expression-type* is operator.

  - *Operand-data-type*: an enumerated variable which indicates the data-type of the operand. It can be **integer**, **boolean**,**port_type_name**, **trait_type_name** **string**. It is null if the *Expression-type* is operator.

- *enabling_condition*: an assertion tree, whose structure is defined earlier.

- *post_condition*: an operation assertion which has same data structure as an assertion tree.

The **Transition-Spec_list** is mandatory in **TROM** class AST.

5.1.11 Time-Constraint

The *Time-Constraint* is represented by a record data structure in the **Time-Constraint_list**. The data structure contains the following data members:

- *time-constraint_name*: a string variable which is not duplicable in the **Time-Constraint_list**.

- *transition_name*: a reference to the existing *transition-spec* in the **Transition-Spec_list.**
- *constrainted_event*: a reference to the existing event in the **Event_list.** *constrainted_event* must be an output event or internal event.
- *lower_bound*: an integer variable which is the lower bound of time period.
- *upper_bound*: an integer variable which is the upper bound of time period and must be greater than or equal to the value of lower bound.
- *state_list*: a reference to the existing **State_list** and can be empty.

The Transition-Spec_list is mandatory in **TROM** class AST.

5.2 The SCS Abstract Syntax Tree Data Structure

The System Configuation Specification input text specifies the instantiation relationship between objects and their classes. It also specifies the aggregation of objects from subsystems. The SCS AST data structure is illustrated in Figure 14.

As shown in Figure 14, the SCS AST is a data construct which is a collection of SCS class member objects. There is a container for each type of class member objects which can be implemented by a linked list. The SCS AST is unique and it can be identified by its SCS-name attribute. Some of the SCS class members such as SCS-name, Port-Link list are mandatory, so their corresponding containers must contain at least one element. Others such as Include list are optional, so their containers can be empty.

5.2.1 General Description for SCS AST class member containers

Except the SCS_name, all the SCS AST class members are represented by a container data type, which can be implemented by a linked-list. To apply the Object-Oriented

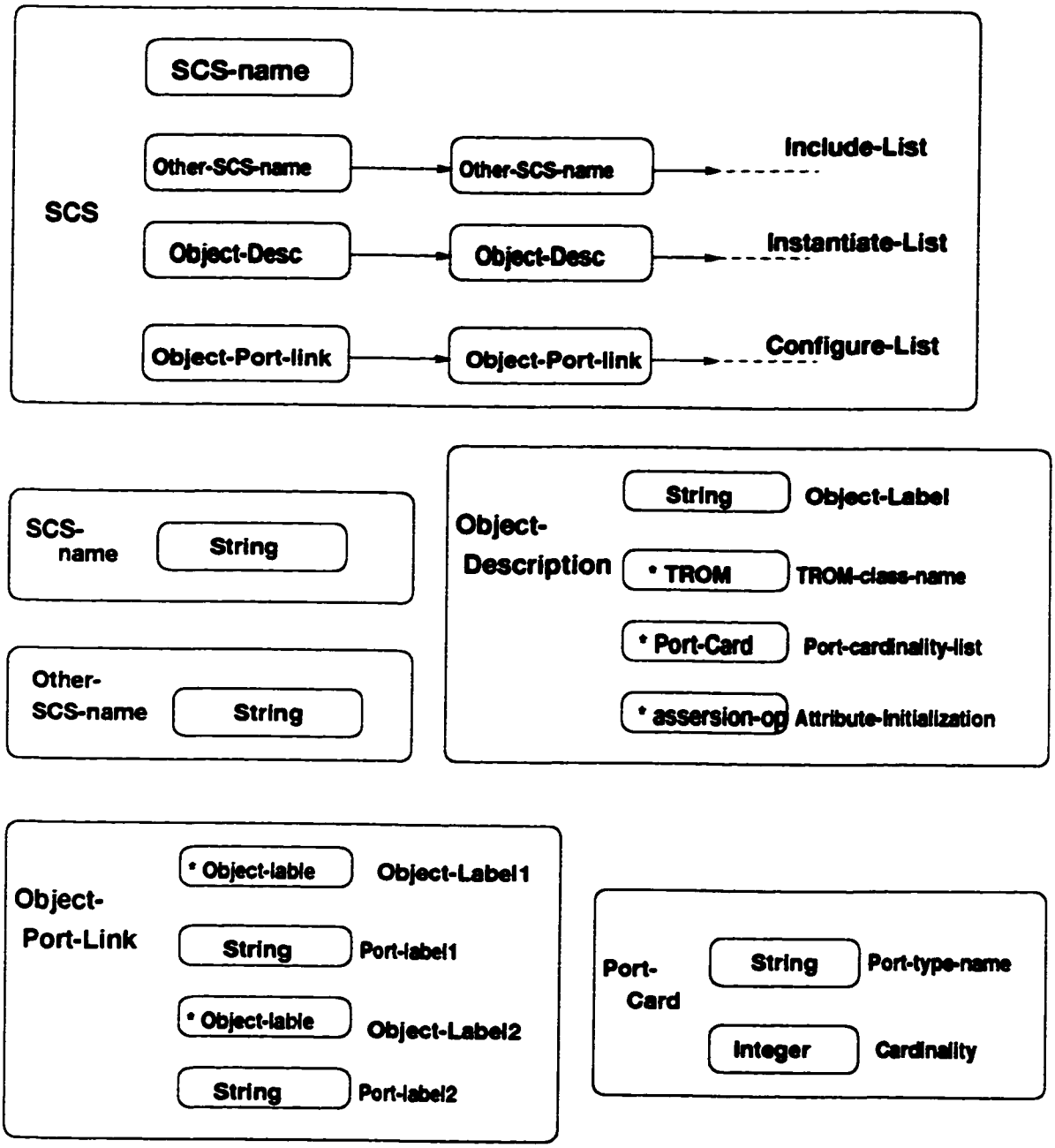


Figure 14: SCS Abstract Syntax Tree Structure.

system design principles, the container is an abstract data type which encapsulates the implementation details. We design the container class which supports the standard access functions as illustrated in **TROM** AST description and also includes the specific function **getPair**, which retrieves an object-port pair from an Object-Port Link container.

5.2.2 SCS name

The *SCS-name* is represented by a string variable which contains an unique SCS identifier. The *SCS-name* is mandatory in SCS AST.

5.2.3 Include

The *Include* is a reference to other existed SCSs which names are not duplicable in the **Include_list**. The **Include_list** is optional in SCS AST.

5.2.4 Instantiate

The *Instantiate* is represented by a data structure in the **Instantiate_list**. The data structure contains the following data members:

- *object_name*: a string variable which is not duplicable in the **Instantiate_list**.
- *TROM_class_name*: a reference to an existing **TROM** *class_name*.
- *port_cardinality_list*: a data structure contains the following data members:
  - *port_type_name*: a reference to an existing **TROM** *port_type_name* in the **Port-type_list**.
  - *Cardinality*: an integer variable to denote the number of ports associated with this port-type.
- *Attribute_Initialization*: a data structure containing the following data members:
  - *attribute_name*: a reference to an existing **TROM** *attribute_name* in the **Attribute_list**.

- *Assertion-op*: an operation assertion which has the same data structure as an assertion tree.

The **Instantiate_List** is optional in SCS AST.

5.2.5 Configure

The *Configure* is represented by a record data structure in the **Configure_List**. The data structure contains the following information:

- *object_name1*: a reference to the existing *Instantiate* object name, says object1, in the **Instantiate_List** from either this SCS AST or other *Included* SCS ASTs.
- *port-name1* : a string variable representing a specific port associated with object1. This *port-name* should be defined the same way as object1.
- *object_name2*: a reference to the existing *Instantiate* object name, says object2 in the **Instantiate_List** from either this SCS AST or other *Included* SCS ASTs.
- *port-name2* : a string variable representing a specific port associated with object2. This *port-name* should be defined the same way as object2.

The **Configure_List** is mandatory in SCS AST.

5.3 The Simulation Event List Abstract Syntax Tree Data Structure

The Simulation Event List AST data structure is illustrated in Figure 15.

As shown in Figure 15, the Simulation Event List AST contains one class member object from its class. The list is unique for each simulation at run time. This list contains references to all objects which are to be simulated.

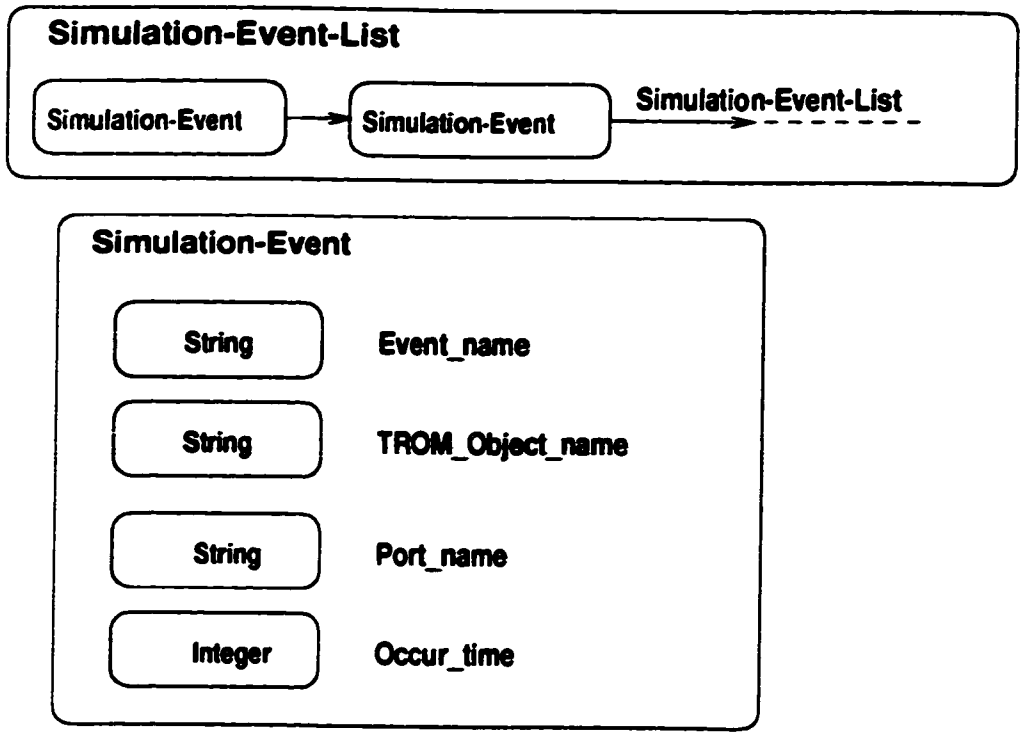


Figure 15: Simulation Event List Abstract Syntax Tree Structure.

5.3.1 Simulation-Event

The *Simulation-Event* is represented by a record data structure in the Simulation-Event_List. The record structure contains the following data members:

- *Event_name*: a reference to an existing *Event_name* in the **Event_List** from a **TROM** in a specified SCS.
- *TROM-object-name* : a reference to an existing *object_name* in the **Instantiate_List** from a SCS.
- *port_name*: a string variable to indicate the specific port which the event will occur. It should be defined in the **Instantiate_List** from a SCS.
- *occur_time*: an integer variable representing the specific time at which the event will occur.

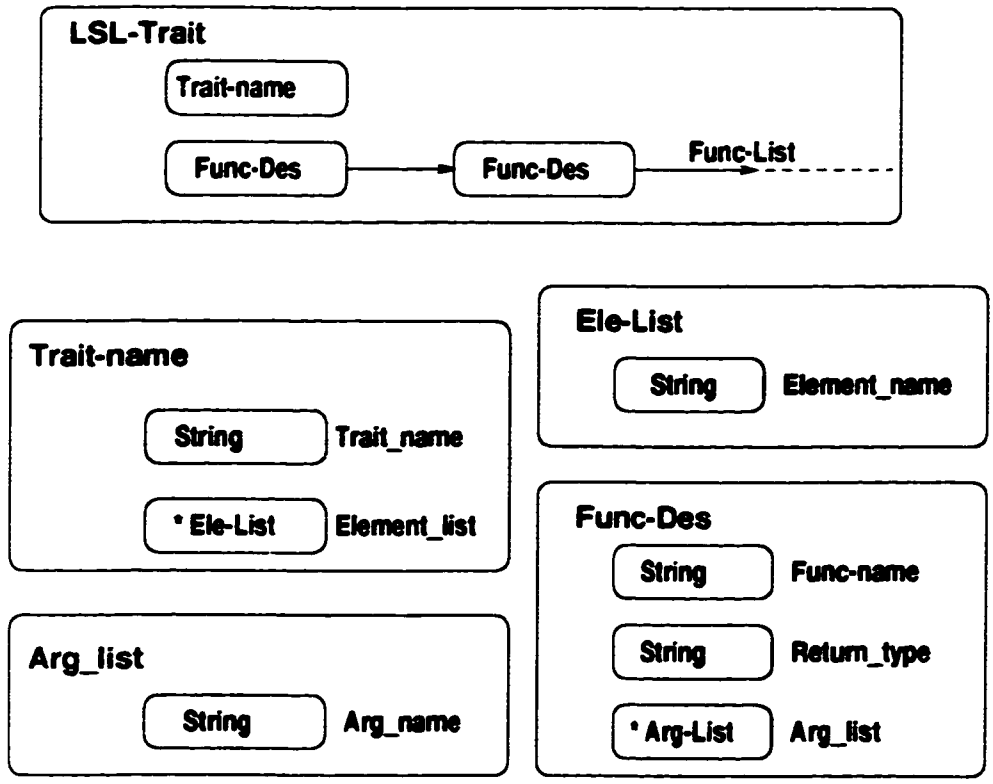


Figure 16: LSL Trait Abstract Syntax Tree Structure.

5.4 The LSL Trait Abstract Syntax Tree Data Structure

The **LSL** Trait input text file could either come from an **LSL** library or user's written trait. The **LSL** Trait AST should be established before other ASTs and their data structures are constructured.

As shown in Figure 16, the **LSL** Trait AST is a data construct which is a collection of its functions. There is a container for each function which can be implemented by a linked list. The **LSL** Trait AST is unique and it can be identified by its Trait-name attribute.

5.4.1 General Description for LSL Trait AST class member containers

All members of the **LSL** Trait AST are represented by a container data type, which can be implemented by a linked-list. The **LSL** Trait AST is only used for semantic analysis for **TROM** and is not accessed during simulation.

5.4.2 Trait name

The data structure of *Trait-name* contains the following data members:

- *Trait-name*: a string variable which contains an unique **LSL** Trait identifier.
- *Element_list*: a container of string variables which represents the parameters associtated with the trait name.

The *Trait-name* is mandatory in **LSL** Trait AST.

5.4.3 Func-Decs

The *Func-Decs* is represented by a data structure in the **Func_List**. The data structure contains the following data members:

- *Func-name*: a string variable which is a unique identifier to the function in the trait.
- *Return_type*: a string variable indicates the return type of the function.
- *Arg_list*: a container of string variables which represents the arguments associtated with this function.

The *Func-Decs* is mandatory in **LSL** Trait AST.

5.5 Larch/C++ specification

We use Larch/C++ to specify the bag(container) and binary tree operations on the AST structure. We first introduce Larch/C++ through the Set example. Figure 17 shows a Larch/C++ interface specification for IntSet, a class which implements set of intergers[10]. For each IntSet operation, the specification consists of a *header* and a *body*. The header specifies the name of the operation, the names and types of the parameters, as well as the return type, and uses exactly the same notation as in C++. The body of the specification consists of an **ensure** clause as will as optional **requires** and **modifies** clauses.

For each IntSet operation, the **requires** and **ensures** clauses specify the pre- and post-conditions respectively. The identifier *self* in the pre- and post-condition assertions denotes the object which receives the message corresponding to the specified method. The **modifies** clause lists those objects whose value may change as the result of executing the operation. Hence, for example, *add* and *remove* are allowed to change the state of an IntSet object but *size* and *isIn* are not. An omitted **requires** clause is interpreted to mean "**requires** *true*" and an omitted **modifies** clause is interpreted to mean that no object is modified by the corresponding method(neither *self*, nor any parameter objects). The link between the IntSet interface specification and the SetTrait LSL specification is indicated by the clause **uses** *SetTrait (IntSet for Set, int for E)*. The *used trait* IntSet provides the names and meaning of the operators {}, *insert*, *delete*, *member*, *size*, and *isEmpty* as well as the meaning of the equality symbol, =, which are referred to in the pre- and post-conditions of IntSet's method specifications. The uses clause also specifies the *type* to *sort* mapping which indicates which abstract values the objects involved in the specification(e.g. *self* and parameter objects) can range over. For example, the abstract values of **IntStack** objects are represented by terms of the sort **Set**.

Associated with each member function specification is the predicate over two states,

```
class IntSet
{
      uses IntSetTrait(IntSet for S):
 public:
  IntSet()
  {
    modifies self;
    ensures self' = {};
  }
  ~IntSet()
  {
    modifies self;
    ensures trashed(self);
  }
  int size()
  {
    ensures result = size(self^);
  }
  void add(int i)
  {
    modifies self;
    ensures self' = insert(i, self^);
  }
  void remove(int i)
  {
    requires ¬isEmpty(self^);
    modifies self;
    ensures self' = delete(i, self^);
  }
  bool isIn (int x)
  {
    ensures result = member(x, self^);
  }
  bool isEmpty(int x)
  {
    ensures result = isEmpty(self^);
  }

}
```

Figure 17: Larch/C++ Specification for for Set.

$$\text{Pre} \longrightarrow (\text{Modifies} \wedge \text{Post})$$

where Pre and Post are the assertions in the **requires** and **ensures** clauses, respectively, and Modifies is the implicit assertion associated with the **modifies** clause. The clause **modifies** $x_1, \ldots, x_n$ implicitly asserts that the method changes the value of no object in the environment of the caller except possibly some subset of $[x_1, \ldots, x_n]$.

It is important to note the following points about a Larch/C++ class interface specification:

- *self* is an abbreviation for (**this*). In C++, *this* represents a pointer to the receiving object so that self = (**this*) is a name representing the receiving object itself.
- A distinction is made between an object and its value by using a plain object identifier(e.g. s) to denote an object, and a superscripted object identifier(e.g. s′ or s^) to denote its value in a state.
- The operators ^ and ′ are used to extract values from objects. An object identifier superscripted by ^ denotes an object's initial value and an object superscripted by s′ denotes its final value. This is similar to the use of superscripts and decorations in VDM and Z. Thus, the assertion self′ = self^ says that the value of the object **self** is left unchanged.
- The header of a Larch/C++ member function specification is deliberately chosen to be exactly the same as C++ member function prototypes.
- The **modifies** clause is an assertion whose meaning is given by considering it to be conjoined to the postcondition. It is syntactically separated from the postcondition to highlight a procedure's potential side effect on the values of objects. It is an example of a *special assertion*. Each Larch interface language comes equipped with its own set of special assertions. For example, in Larch/C and Larch/C++, there is a keyword **trashed** which is used to indicate

deallocation of component objects in the destructor of a class. These special assertions can be regarded as syntactic sugar for first-order assertions about state.

Below are given the interface specifications for Bag(container class) Bag Iterator, Binary Tree and Binary Tree Iterator figures. We refer the reader to the reports[3] for the specifications of LSL traits used in these specifications and other included interface specifications. Note that *any* means both pre and post states. The specifications capture the operations in the AST structure explained in Section 5.1.

```
imports HashDictionary, CollectableInt;
class Bag : virtual public Collection
{
uses Bag_hashdict(Collectable for E, Bag for HT);
public:
Bag(unsigned N = DefaultCapacity )
 {
  contructs self;
  ensures self' = create(N);
 }
virtual Collectable* find(const Collectable* e1) const
 {
  ensures ∀i : item(if i ∈ self\any ∧ ((i.key!any) = ((*e1)\any))
   then *result = i.key else result = NULL);
 }
virtual Collectable* insert(Collectable* c)
 {
  modifies self;
  ensures ∀i : item(if i ∈ self^ ∧ ((i.key!pre) = (*c)^)
   then *result = i.key ∧ (valueatkey(i.key, self')!post) =
    (valueatkey(i.key, self^)!pre) + 1
   else result = c ∧ self' = insert(givesitem(c, 1), self^));
 }
virtual Boolean isEmpty() const
 {
  ensures isEmpty(self\any);
 }
virtual Collectable* remove(const Collectable* a)
 {
  modifies self;
  ensures ∀i : item(if i ∈ self^ ∧ ((i.key!any) = (*a)\any)
   then remove(i, self^) = self' ∧ *result = i.key else result = NULL);
 }
virtual void removeAndDestroy(const Collectable* target)
 {
  modifies self;
  ensures ∀i : item((i ∈ self^ ∧ ((i.key!any) = (target*)\any)) ⇒
   (if valueatkey(i.key, self^) = 1
   then self' = remove(i, self^) ∧ trashed(i.key) ∧ trashed(i.value)
   else (valueatkey(i.key, self')!post) = (valueatkey(i.key, self^)!pre) − 1));
 }
};
```

Figure 18: Larch/C++ Specification for Bag.

```
imports Iterator, Bag;
class BagIterator : public Iterator
{
uses BagIterObj(BagIterator for BagIterObj, Bag for HT, Collectable for E);
public:
BagIterator(const Bag& h)
 {
  contructs self;
  ensures self'.Iterator = create(map(h\any,0))∧
   ∀i : item(∃i1 : item(i ∈ h\any ⇒
   FindOnList(map(h\any,0),i1) ∧ i = i1)) ∧ self'.count = 0;
 }
virtual Collectable* operator()()
 {
  modifies self;
  ensures self' = MoveBagIterator(self^,1,any)∧
   (if ItemAt(self'.Iterator) = UNDEFINED
   then result = NULL
   else (*result) = ItemAt(self'.Iterator).key);
 }
virtual Collectable* findNext(const Collectable* target)
 {
  requires notNull(target);
  modifies self;
  ensures self' = NextItemInBag(self^,(*target),any)∧
   (if ItemAt(self'.Iterator) = UNDEFINED
   then result = NULL
   else (*result) = ItemAt(self'.Iterator).key);
 }
};
```

Figure 19: Larch/C++ Specification for BagIterator.

```
imports Collection;
class BinaryTree : virtual public Collection
{
uses BinaryTree(BinaryTree for Bin[Obj[E]], Collectable for E).ClassID;
public:
BinaryTree()
 {
  contructs self;
  ensures self' = Empty;
 }
¬ BinaryTree()
 {
  modifies self;
  ensures trashed(self');
 }
virtual Collectable* find(const Collectable* target) const
 {
  requires notNull(target);
  ensures ∀e : Obj[Collectale]
   (if FindOnTree(self\any, e) ∧ e\any = (*target)\any
   then (*result) = e
   else result = NULL);
 }
virtual Collectable* insert(Collectable* c)
 {
  requires notNull(c);
  modifies self;
  ensures self' = (AddNodeValue(self^, (*c)) ∧ result = c);
 }
virtual Boolean isEmpty() const
 {
  ensures result = IsEmpty(self\any);
 }
virtual Collectable* remove(const Collectable* target
)  {
  requires notNull(target);
  modifies self;
  ensures ∀i : Obj[Collectable]
   (if FindOnTree(self^, i) ∧ i\any = (*target)\any
   then self' = DeleteNode(self^, i) ∧ (*result) = i
   else unchanged(self) ∧ result = NULL);
 }
};
```

Figure 20: Larch/C++ Specification for BinaryTree.

```
imports BinaryTree, Iterator;
class BinaryTreeIterator : public Iterator
{
uses BinIteratorObj(BinaryTreeIterator for BinIter[Obj[E]],
  BinaryTree for Bin[Obj[E]], Collectable for E);
public:
BinaryTreeIterator(const BinaryTree& b)
 {
  contructs self;
  ensures self' = create(map(b\any)) ∧ ∀e : Obj[Collectable]
   (FindOnTree(b\any, e) ⇒ ∃e1 : Obj[Collectable]
   (FindOnList(map(b\any), e1)));
 }
virtual Collectable* operator()()
 {
  modifies self;
  ensures (self' = MoveIterator(self^, 1))∧
   (if ItemAt(self') = UNDEFINED
   then result = NULL
   else (*result) = ItemAt(self'));
 }
virtual Collectable* findnext(const Collectable* target)
 {
  requires notNull(target);
  modifies self;
  ensures self' = NextEqualItem(self^, (*target)\any, pre)∧
   (if ItemAt(self') = UNDEFINED
   then result = NULL
   else (*result) = ItemAt(self'));
 }
};
```

Figure 21: Larch/C++ Specification for BinaryTreeIterator.

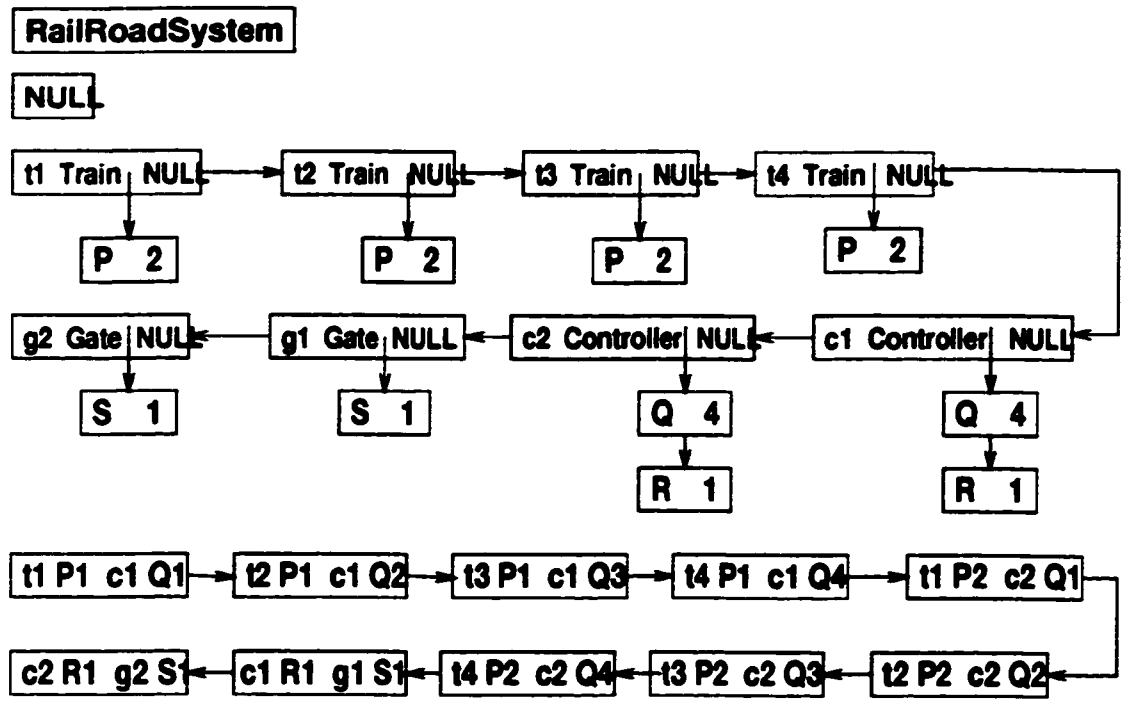


Figure 22: AST for Subsystem - Trait-Gate-Controller System.

5.6 Abstract Syntax Tree for Train-Gate-Controller Example

The ASTs for the **TROM** class in Train-Gate-Controller Example are shown in Figure 23 and 24. Figure 22 gives the Subsystem AST for Train-Gate-Controller Example. These representations are constructed by the interpreter after the input specifications are found to be syntactically correct.

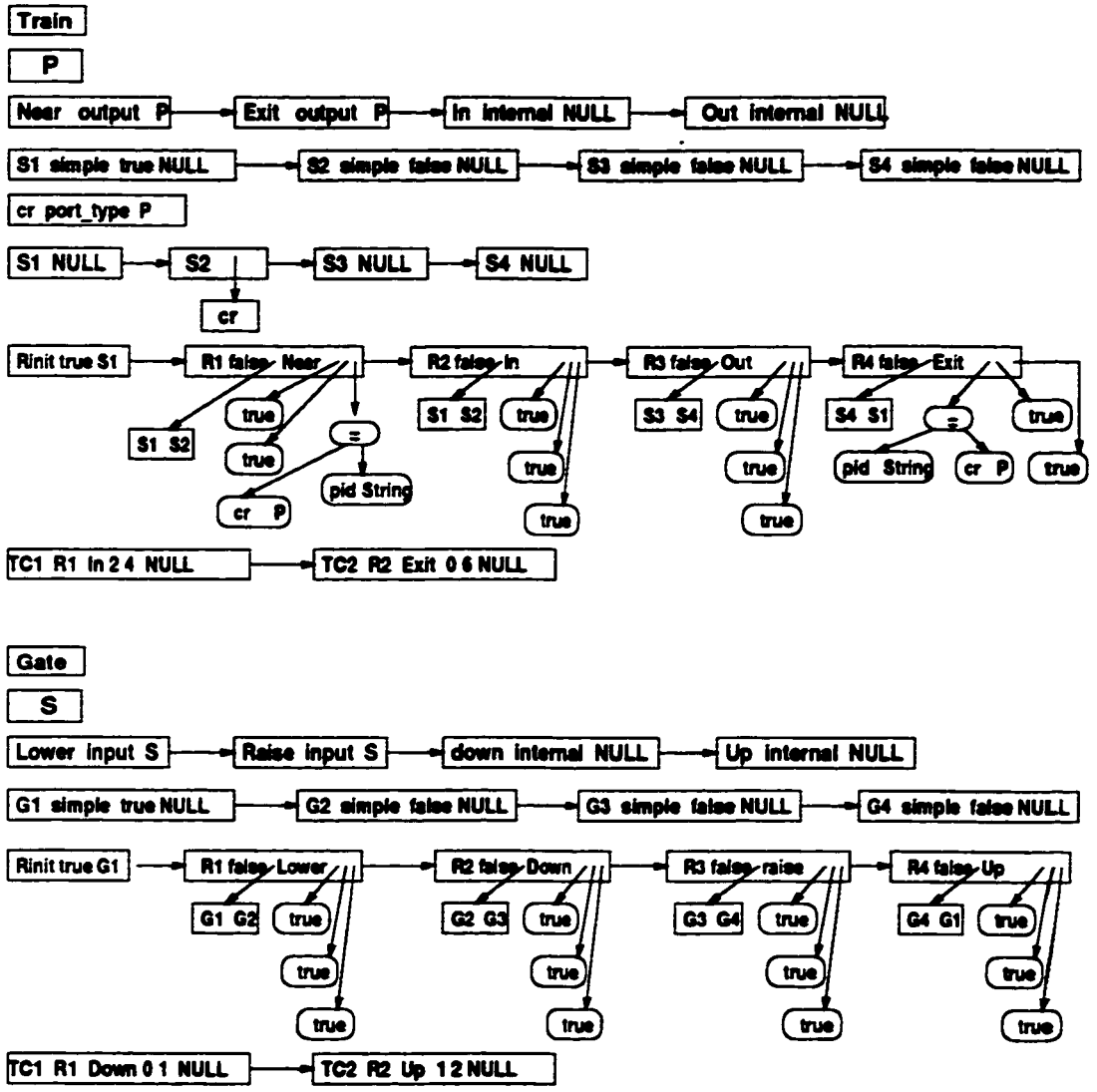


Figure 23: Abstract Syntax Tree for **TROM** class - Train, Gate.

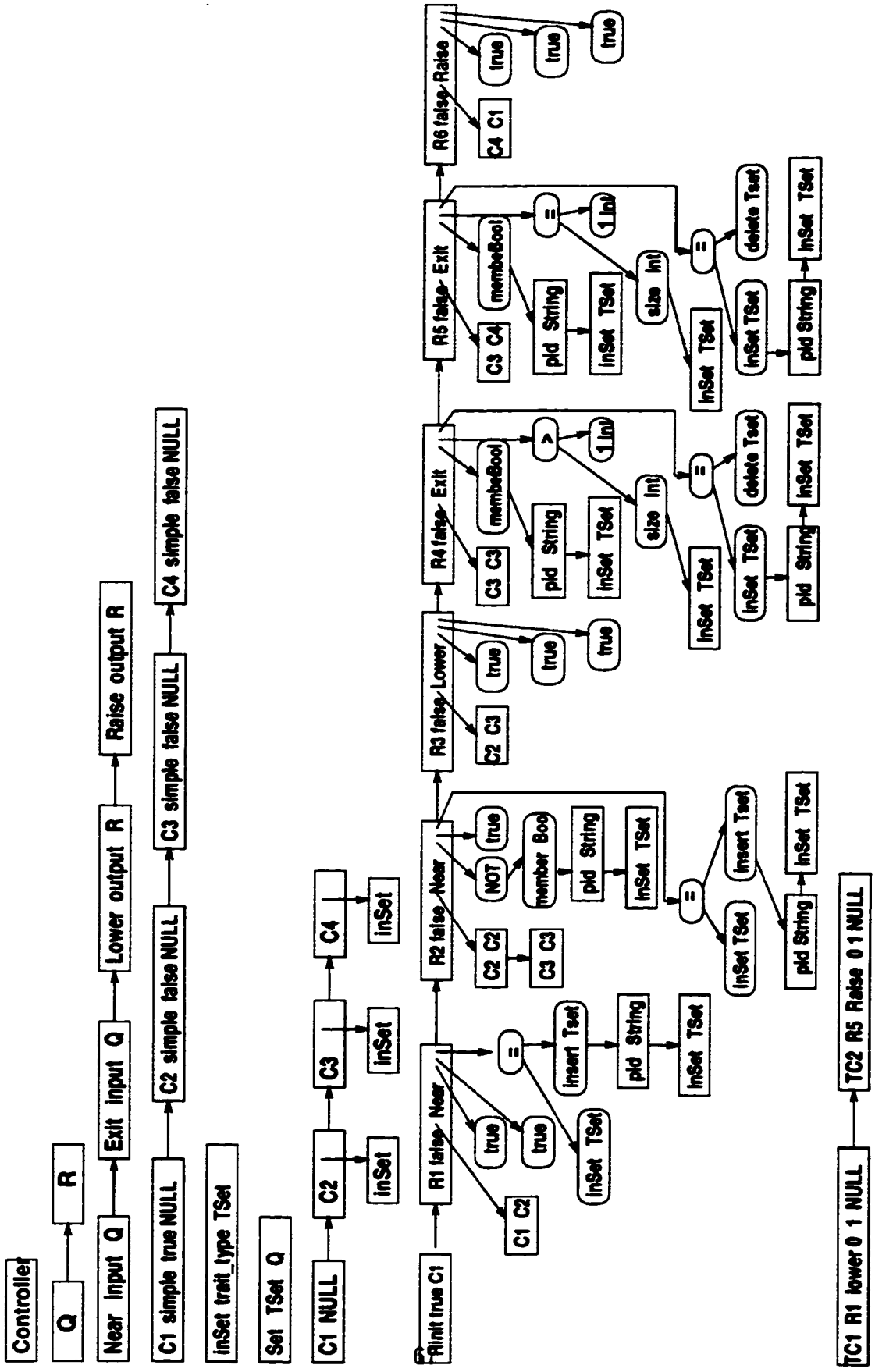


Figure 24: Abstract Syntax Tree for **TROM** class - Controller.

Chapter 6

Semantic Analysis for TROM and Subsystems

In order to use this simulation tool to actually simulate a given system, users must input syntactically and semantically correct **TROM** classis and Subsystems. In Chapter 4, we discussed a syntax analyses for them. This chapter will focus on semantic analysis. We identify the properties to be checked and explain how AST is to be used in this analysis. We also give the result of inputting to the semantic analyzer an example which is syntactically correct, but semantically incorrect. The errors generated by the semantic analyzer help user locate the errors in input specifications.

6.1 What properties need to be checked?

6.1.1 TROM class semantic analysis

1. Each **TROM** class has its unique name. That means if there are two **TROM** classes with the same name, an error message " Duplicate **TROM** class name!" will be given.

2. *Port_types* listed in **TROM** class header should not be duplicated.

3. The input, output and internal events should be disjoint. Any *port_type* associated to an input or output event must be pre-defined within this **TROM** class header. Otherwise, the error message " Event referred to an undefined

port type!" will be given. The internal events are not associated with any port_type.

4. A **TROM** has at least two states. If a state is *complex*, there should be exactly one starting state and at least one other state in the decompositions of a state. That means one state alone is not allowed in the decompositions of a state.

5. An *attribute* can be only one of the following two types:
   i) an abstract data type supporting a data model.
   ii) a port reference type.
   The *port_type* used with each attribute must be pre-defined in the **TROM** class. The *trait_type* is a type of an attribute which is an abstract data type used in the **LSL** trait.

6. Each *trait_name* used in a **TROM** class must exist in **LSL** traits library(including the input file) and its arguments can only be *port_type* or *trait_type* which must be pre-defined within this **TROM** class.

   For example, if a **TROM** class uses one Set trait which is not included in the input file, the error message "Trait is not defined!" will show up. Also, if the Set exists but the arguments are not either defined in port_type list in the class or trait_type in attribute, error messages "... refer to an undefined port_type" or " ... refer to an undefined trait_type" will be given. That is, the name of the trait, and its arguments(the number of arguments and their types) should match.

   It is noticeable that our system is case sensetive. It means that if an **LSL** trait name is set and a trait name within a **TROM** is Set, an error message is given.

7. In *Attribute-function*, states must belong to state set in this **TROM** class. Attributes must belong to attribute set in this **TROM** class. The set of attributes can be empty.

8. In *Transition-Spec*, the name must be unique within one **TROM** class.

   - If there is an initial transition specification, it has only source state and no event associated with it. The initial transition is optional and only one is allowed within a **TROM** class.
   - The *state_pair* can be different states and can also be same state. States in transition specification should belong to the state set in this **TROM** class.
   - Every attribute in transition specification should belong to the attribute set in this **TROM** class.
   - *Port-condition* is an assertion, which only associates with external event. That means internal event doesn't have port_condition. Any port-type used in the port condition should be a valid *port_type* as defined in the class header.
   - Functions invloved in the *port_condition*, *enab_condition* and *post_condition* must be defined in the corresponding **LSL** trait which is linked by the *trait_name* in this **TROM** class.

9. In *Time-constraint*, only output and internal events can have time constraints associated with them.

   The value of *max* must be greater than or equal to the value of *min*. Any *tran_spec_lable*, *event_name* and *state_name* used here must be pre-defined within this **TROM** class.

6.1.2 Subsystem semantic analysis

1. SCS name should be unique in a system . The error message will show up when there are identical SCS names.

2. Each SCS name listed in the *includes* clause should be defined in the system and their associated **TROM**s should also be defined within the system.

3. There exists multiple inheritance in the *Include* clause. For example, SCS system1 may include system2 and system2 may include system3 and so on. This should not cause a deadlock. For example, in the above example, if system1 appears in the *Include* list of system2, it will cause deadlock. Our interpreter can detect such definitions and give an error message to the user.

4. In the *Instantiate* clause, all names of instantiated objects in the system must be unique. Each object can be instantiated from this SCS subsystem or from other included subsystem **TROM** class.

   - The **TROM** class which instantiates an object must be defined previously in the system otherwise an error message will show up.
   - The *port_type* associated with the instantiated object should be valid in that specific **TROM** class.

5. In the *Configure* clause, every pair of object and port should be valid in the system:

   - Each *object_name* in a pair should exist from the *Instantiate* clause either within this subsystem or the *Include* subsystems.
   - The port-type of a *port_name* following each object name must be defined within the same *Instantiate* clause as the object and the number of a port should be valid in the sense of not exceeding the port cardinality of the corresponding *Instantiate* clause for this object. An error message will be given if this does not hold.

6.2 Implementation of the Interpreter

We have chosen C++ as the language of implementation for the Interpreter. It is chosen because our design is based on object-oriented methodology and C++ can fully support it. The parsing is done using **Flex** and **Bison**, provided by UNIX system. The parse program which is using **Flex** and **Bison**[7] is listed in Appendix A.

The parse generated by **Bison** calls the lexer(using **Flex**) *yylex*()- which is the main function in the lexer–whenever it needs a token from the input file. **Bison** also generates a file called **y.tab.cc** which is the C++ language parser that can be linked with other C++ programs.

The advantage of using **Flex** and **Bison** is that it can generate tokens automatically against the predefined grammar and reduce a lot of workload of hand writting parse program. The disadvantage of **Flex** and **Bison** is that the program will stop at any time if there is a parse error and the programmer does not have any control on it.

If the input specification is parsed correctly, then the AST as described in Chapter 5 corresponding to the specification is built. However, semantic analysis, as explained in this section must be done to ensure the correctness of the internal representation of **TROM**s. Since the programmer has full control during semantic analysis, an error file may be generated for debugging the design.

The Interpreter is running under UNIX system.

6.3 How AST is used in Semantic Analysis?

As we explained the AST structure in Chapter 5, there are only two types of data structures used in the implementation: Linked list and Binary tree. After building an AST, all information are stored in different linked lists and the container is the base to hold all pointers for these lists.

The C++ definition of the AST is listed in Appendix B.

Several semantic routines are used by the Interpreter and we can't explain all their algorithms. We chose some typical ones follow:

1. **Check duplicate names**

It can be done on the fly during parsing before the token gets inserted into an AST. For example, **TROM** class name can't be duplicated. Let P be the pointer to all-**TROM** list. When parser gets a **TROM** class name T1, it will do the following:

```
for each P ≠ NULL
    get a TROM class name T2 from all-TROM list
    if T1 = T2 then
       an error message is given
       break
    else move P to the next
    endif
endfor
insert T1 into the all-TROM list
```

2. Check Trait in **TROM** class specification

   This analysis has two purposes: one is to check whether or not there is the same name **LSL** trait in this input specification and the other is to ensure consistency arguments. For example, if there is a trait named Set in a **TROM** class specification but the **LSL** trait Set cannot be found in library in the input file, an error message will be given. That means, user cannot use an undefined **LSL** trait to specify his system behavior.

   Another purpose is to build up a symbol table to acctually give the type of arguments for this particular **TROM** class in the sense of doing type checking for Transition specification assertions later on. For example, we have Set(e, S) trait in **LSL** Library. In Controller **TROM** class specification, we use *Set*([*@Q*, *TSet*]) trait to describe the object behavior. The Interpreter will link this two tiers and generate a symbol table which for each e in **LSL** trait Set, will be substituted by *@Q*, and each S will be substituted by *inSet* which is the name of an attribute corresponding to *TSet*.

   This is essentially linking the two tiers – Lower tier(**LSL**) and Middle tier(**TROM**).

Let P be the pointer to all-**LSL** list. For each trait name T1 in a **TROM**. the method is the following:

```
for each P ≠ NULL
    get the LSL trait name T2 from all-LSL list
        if T1 = T2 then
           if number of arguments of T1 = number of arguments of T2
           then build up a symbol table
           else an error message is given
           break
           endif
        else move P to the next
        endif
endfor
an error message is given
```

3. Check **LSL** function in Transition Specification

   The **LSL** functions can only exist in port-condition, pre-condition and post-condition assertions. For example, a function *member*(**pid**, *inSet*) may occur as part of an enabling condition of a transition. For semantic analysis, we take the argument *inSet*, which is an attribute name, and go to the attribute list to find out the type of this attribute. Then, from the type of the attribute, which is *TSet*, we can find out the trait name which is Set. After that, we can access the symbol table for Set to find out whether *member* is the valid function in it. The number of arguments and their type for *member* should match. Otherwise, the interpreter will give error messages. Note that **pid** is a legal argument for any port-type. For each **LSL** function L,The method is the following:

```
get the last argument A1 of a function
for each attribute type A2 in attribute list do
```

```
if A1 = A2 then
   for each trait T2 in trait list do
       if the last argument of T2 = A1 Then
       goto symbol table of T2 serching L
            if L is found then
            check the consistency of arguments of L in the table
                 if match then
                 break
                 else give error message
                 endif
            else give error message
            endif
       else give error message
       endif
   endfor
endfor
```

4. Check instantiated object from Instantiate clause in Subsystem Specification

   An object from a **TROM** is instantiated in subsystem specification.
   For example, $t1$ :: *Train*[@P : 2];
   means that $t1$ is an object from *Train* class and it has two ports which are P1 and P2 of port-type @P. First, the object name has to be legal(no duplicate) which could be checked during syntax analysis. Then we check whether *Train* is an existing class in the AST. If *Train* exists, then we check whether port-type @P is valid in the *Train* class. Otherwise, an error message will be given. Having done this, the instantiated object is ensured to be correct. Then, we will build a template for each object according to its ports such as: $t1.P1$ and $t1.P2$. It will be used when we check the object-port configuration later on.

Note that this stage is essentially linking the two tiers - Top tier(Subsystem) and Middle tier(**TROM**). Obviously, higher tier uses lower tier definition so that there exists multi-dependencies in the design specification.
The method to check an instantiated object is the following:

```
for each object in Instantiat list do
    get a TROM class name T from the object
    if T is found in all-TROM list then
       get a port-type @P from the object
       if @P is found in T then
          build template for links
       else give error message
       endif
    else give error message
    endif
endfor
```

5. Check object-ports link from Configure clause in Subsystem Specification
   The Configure clause defines system/subsystem by aggregating objects specified in the Instantiate clause and other subsystem specifications imported through the Include clause.
   For example, *t1.@P1 < − > c1.@Q1* simply means that port *P1* of object *t1*(from Train class) is linked with port *Q1* of object *c1*(from Controller class). In fact, it defines the interaction between two objects and the way it links. To check the correctness of the linking, we take the first object name *t1* to go to the Instantiate clause and check whether it exists. If it is not found in this Instantiate clause, we check other SCS Instantiate clause through the *Include* clause. Having found *t1*, we then check whether or not the port *P1* is valid [that is *P* should be pre-defined within the object and *1* should be the valid number(not to exceed the number within the instantiated object)]. Same procedure checks

the second object and its port. Error messages will be given if any mismatch is detected.

Having done this, we only ensure that these two pairs of objects and ports are valid. Next, we need to check whether or not the link between those two objects is valid. That is, one port for an object can be linked only once with a port of another object. That means the template for an object can only be assigned once. Otherwise, there will be a collision and an error message will be given. For example, consider the following configure specification:

t1.@*P1* < − > *c1*.@*Q1*;

t1.@*P1* < − > *c2*.@*Q1*;

The Interpreter will detect the wrong link between *t1* and *c2*. The method to check the linking is the following:

```
for each object O in Configure list do
    if O is found in this SCS Instantiat list then
    A: get the port P_n of object O
       if P is also found in this SCS Instantiat list within O then
          if n <= the number in the SCS Instantiat list within O then
          check template for collision of link
                if no collision then break
                else give error message
                endif
          else give error message
          endif
       else give error message
       endif
    else get Instantiate list from other included SCS
    repeat checking from A
    endif
give error message
```

```
endfor
```

6.4 Semantic Analysis Report Example

To illustrate the error detection capabilities of the semantic analyzer, we give an example below.

```
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


Class Train [@P, @P]
  Events: Near!0, Exit!P, In, Out
  States: *S1, S2, S3, S4
  Attributes: cr:@P
  Attribute-function:S1 -> {}; S5 -> {}; S3 -> {}; S2 -> {tr};
  Transition-Spec:
        Rinit: <S1>;Create();
        R1: <S1,S2>; Near(true); true => cr' = pid;
        R2: <S2, S3> ; In; true => true;
        R3: <S3, S4> ; Out; true => true;
        R4: <S4,S1>; Lower(cr=pid); true => true;
```

```
  Time-Constraints:
        TC1: (R5, In, [2,4], {});
        TC1: (R1, Exit, [0,6], {});
end

Class Gate [@S]
  Events: Lower?S, Raise?S, Down, Up
  States: *G1, G2, G3, G4
  Transition-Spec:
        Rinit: <G1>;Create();
        R1: <G1,G2>; Lower(true); true => true;
        R2: <G2, G3> ; Down; true => true;
        R3: <G3, G4> ; Raise(true); true => true;
        R4: <G4,G1>; Up; true => true;
  Time-Constraints:
        TC1: (R1, Down, [0,1], {});
        TC2: (R3, Lower, [1,2], {});
end

Class Controller [@Q, @R]
  Events: Near?Q, Exit?Q, Lower!R, Raise!R
  States: *C1, C2, C3, C4
  Attributes: inSet:TSet
  Traits: Set[@Q, TSet]
  Attribute-function: C1 -> {}; C2 -> {inSet}; C3 -> {inSet}; C4 -> {inSet};
  Transition-Spec:
        Rinit: <C1>;Create();
        R1: <C1,C2>; Near(true); true => inSet' = insert(inSet, pid);
        R1: <C2, C2>,<C3, C3> ; Near(NOT(belong(pid, inSet)));
              true => inSet' = insert(pid, insert(pid,inSet)) ;
        R3: <C2, C3> ; Lower(true); true => true;
```

```
        R4: <C3, C3> ; Exit(belongto(pid, inSet)); (size(inSet)>1) =>
                        inSet' = delete(pid, inSet) ;
        R5: <C3, C4> ; Exit(belongto(pid, inSet)); (size(inSet)=1) =>
                        inSet' = delet(pid, inSet) ;
        R6: <C4,C1>; Raise(true); true => true;
  Time-Constraints:
        TC1: (R1, Lower, [0,1], {});
        TC2: (R5, Raise, [0,1], {});
end

SCS RailRoadSystem
  Instantiate:
        t1:: Trai[@P: 2];
        t2:: Train[@Q: 2];
        t3:: Train[@P: 2];
        t4:: Train[@P: 2];
        c1:: Controller[@Q: 4, @R:1];
        c2:: Controller[@Q: 4, @R:1];
        g1:: Gate[@S:1];
        g2:: Gate[@S:1];
  Configure:
        t1.@P1 <-> c1.@Q1; t1.@P2 <-> c2.@Q1;
        t2.@P1 <-> c1.@Q2; t2.@P2 <-> c2.@Q2;
        t3.@P4 <-> c1.@Q3; t3.@P2 <-> c2.@Q3;
        t4.@P1 <-> c1.@Q4; t4.@P2 <-> c2.@Q4;
        c1.@R1 <-> g4.@S1; c2.@R1 <-> g2.@S1;
end

SEL: Near,t1,P2,2;
     Ext,c1,Q1,4;
```

This example is syntactically correct but has a number of semantic inconsistencies. Our analyzer detects these inconsistencies and gives the following error messages to user.

```
Duplicated port type: P at line no. 14
Duplicated time-constraint: TC1 at line no. 28
Duplicated Transition-spec: R1 at line no. 58

Semantic Checking for TROM Class: Train
Event 'Near' referred to an undefined port type 'O'
Undefined state 'S5' in Attribute-function
Undefined attribute 'tr' in Attribute-function
Undefined trigger event 'Lower' in Transaction-spec 'R4'
Time-constraint 'TC1' refers to an undefined transition 'R5'

Semantic Checking for TROM Class: Gate
Time-constraint 'TC2' refers to an input event 'Lower'
Semantic Checking for TROM Class: Controller
Invalid arguments for LSL function 'insert'
Invalid post-condition in Transaction-spec 'R1'
Undefined LSL function 'delet'
Invalid post-condition in Transaction-spec 'R5'

Semantic Checking for SCS: RailRoadSystem
Object 't1' is instantiated from undefined TROM class 'Trai'
SCS instance 't2' refers to an undefined port_type 'Q'
 (in TROM class 'Train')
Undefined port type 'P' of port 'P1' in Configure:
 t2.@P1<->c1.@Q2
Undefined port type 'P' of port 'P2' in Configure:
 t2.@P2<->c2.@Q2
```

```
Invalid port# '4' of the port 'P4' in Configure:
 t3.@P4<->c1.@Q3
Undefined object instance 'g4' in Configure:
 c1.@R1<->g4.@S1

Undefined TROM class 'Trai' in Simulation Event 'Near'
Invalid Simulation Event 'Ext'
```

Chapter 7

Axiom Generator

In [1] the semantics of **TROM** is expressed through a set of axioms in a first order temporal logic. The axioms provide a framework within which the requirements properties of a reactive system can be verified against the system design described by the **TROM**s. The goal of this Chapter is to describe the Axiom Generator module, which prepares the axioms of **TROM**s in a system design for use by the verifier in the animator.

7.1 TROM Axiom System

The first order temporal logic uses the three syntactically higher-order predicates *Hold*, *HoldFor*, and *Occur* and the temporal relationships among time intervals expressed by the predicates shown in Figure 25.

In addition, the two other predicates used for expressiveness are

$$Meet(T1, T2, T3) = Meet(T1, T2) \wedge Meet(T2, T3)$$
$$In(T1, T2) = During(T1, T2) \vee Starts(T1, T2) \vee Finishes(T1, T2)$$

The predicate *Hold* is used to express properties that can hold or not hold during a finite interval. For example, $Hold(A.\theta, T)$ asserts that **TROM** A is in state θ in the interval of time T and in every subinterval of T. The predicate $HoldFor(A.\theta, T)$ is true when T is the maximal interval for which $Hold(A.\theta, T)$ is true.

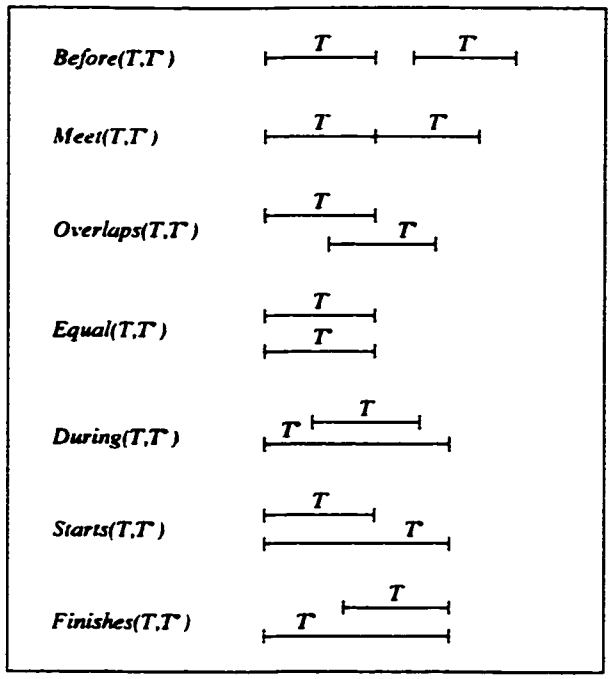


Figure 25: The predicates defining temporal relationship between intervals

The *Occur* predicate is used to express event occurrences in the system. For example, the predicate *Occur*($A.e,p_i,t$) states that the event e occurs at port p_i in TROM A during an *elementary* interval of time. The interval t is to regarded as "very small" compared to T, so that during the occurrence of an event A **TROM** is in the source state of the transition involving the event. Whenever the context is clear we omit the reference to **TROM** in the predicates.

The axiomatization consists of two kinds of axioms: general axioms, which are shared with any first order theory with equality and temporal **TROM** axioms and several temporal constraints. For example, the following axioms are general axioms.

$$\forall\, x \bullet (\varphi \longrightarrow \alpha) \longrightarrow (\forall\, x\, \varphi \longrightarrow \forall\, x\, \alpha)$$

$$u = s \longrightarrow (\varphi \longrightarrow \varphi')$$

$$\mathrm{Hold}(\neg\, \theta, T) \leftrightarrow \forall\, t\, [\mathrm{In}(t, T) \longrightarrow \neg\, \mathrm{Hold}(\theta, t)]$$

There are eleven axioms of temporal constraints associated with a **TROM**. Informally, these axioms assert the total behavior of a **TROM**. For example, **TROM**s

must respect atomic-event property, which can be stated as "at any time t, there can be at most one event occurring in a **TROM** and moreover an event can occur only at one port". Clearly, this property is enforced during the semantic analysis stage. However, for formal verification purposes it is essential to state this property in the following logical axioms:

Atomic-event axiom: ..... (AE)

(a) $Occur(e_1, p_i, t) \wedge (e_1 \neq e_2) \longrightarrow \neg$ Occur(e_2, p_j, t)

(b) $Occur(e, p_i, t) \wedge (p_i \neq P_j) \longrightarrow \neg\ Occur(e, p_j, t)$

For the sake of clarity and immediate reference to later sections of this Chapter the other axioms from [1] are reproduced below:

1. *Silent-event axiom* ..... (SE)

$Occur(\mathsf{tick}, o, t) \longrightarrow \forall\ e, p_i \bullet \neg\ Occur(e, P_i, t)$

2. *State-hierarchy axioms* ..... (SH)

(a) $Hold(\theta_i, T) \wedge \theta_i \in \Phi_s(\theta_j) \longrightarrow Hold(\theta_j, T)$

(b) $Hold(\theta_j, T) \longrightarrow \forall_{\theta_i' \in \Phi_s(\theta_j)}\ Hold(\theta_i\ ', T)$

3. *State-uniqueness axiom* ..... (SU)

$Hold(\ \theta, T) \wedge (\theta \neq \theta\ ') \wedge (\theta \notin \Phi_s(\theta\ ')) \wedge (\theta\ ' \notin \Phi_s(\theta)) \longrightarrow \neg\ Hold(\theta\ ', t)$

4. *Initial-state axiom* ..... (IS)

$Hold(\theta_0, \mathrm{t}_{init})$

5. *Initial-attribute axiom* ..... (IA)

$\varphi(\mathrm{t}_{init}) \longrightarrow \varphi(\mathrm{t}_{init})$

6. *Dormant-attribute axiom* ..... (DA)

$$\forall x_i \in \overline{\Phi_{at}}(\theta) \bullet Hold(\theta, t') \wedge Meet(t, t') \longrightarrow x_i(t) = x_i(t')$$

7. *Occurrence axiom* ..... (OC)

$$Occur(e, p_i, t) \longrightarrow \left(\begin{array}{c} Hold(\theta_1, t) \wedge \varphi^1_{en}(t) \wedge \varphi^1_{port}(t)[pid \mapsto p_i] \\ \wedge \\ \vdots \\ \wedge \\ Hold(\theta_n, t) \wedge \varphi^n_{en}(t) \wedge \varphi^n_{port}(t)[pid \mapsto p_i] \end{array} \right)$$

8. *Transition axiom* ..... (TR)

$$Hold(\theta, t) \wedge Occur(e, p_i, t) \wedge Meet(t, t') \longrightarrow Hold(\theta', t') \wedge \varphi_{post}(t, t')[\mathsf{pid} \mapsto p_i]$$

9. *Persistence axiom* ..... (PS)

$$\left(\begin{array}{c} Hold(\theta_1, t) \wedge Meet_n(T, t, t') \wedge \\ (\neg\, Occur(e_1, P_1, t) \vee \ldots \vee \neg\, Occur(e_n, P_n, t)) \end{array} \right) \longrightarrow \left(\begin{array}{c} Hold(\theta, t') \wedge \\ (\forall x_i \in \Phi_{at}(\theta) \bullet x(t) = x(t')) \end{array} \right)$$

10. *Time-constraint axioms* ..... (TC)

(a) *Activation axiom*: ..... (ac)

$$Trigger(e, t_a) \wedge \neg\, Disable(e, t) \wedge Meet(t_a, t) \longrightarrow Enable(e, t_a, t)$$

(b) *Constraint-event axiom*: ..... (ce)

$$Occur(e, p_i, t) \longrightarrow Occur(f, p_j, t_a) \wedge Within(t_a, l, u, t)$$

(c) *Enabling axiom*: ..... (en)

$$Enable(e, t_a, t) \wedge \neg\, Occur(e, p_i, t) \wedge Meet(t, t') \wedge$$

$$\neg\ Disable(e, t') \longrightarrow Enable(e, t_a, t')$$

(d) *Disabling axiom*: ..... (ds)

$$Enable(e, t_a, t) \wedge Meet(t, t') \wedge Disable(e, t') \longrightarrow \neg\ Enable(e, t_a, t')$$

(e) *Firing axiom*: ..... (fr)

$$Enable(e, t_a, t) \wedge Occur(e, p_i, t) \wedge Within(t_a, 0, u, t) \wedge Meet(t, t') \longrightarrow \neg\ Enable(e, t_a, t')$$

(f) *Prohibition axiom*: ..... (ph)

$$Enable(e, t_a, t) \wedge Within(t_a, 0, l, t) \longrightarrow \neg\ Occur(e, p_i, t)$$

(g) *Obligation axiom*: ..... (ob)

$$Enable(e, t_a, t) \wedge \forall\ t'[\ Within(t_a, 0, u, t') \longrightarrow \neg\ Disable(\ e, t')] \longrightarrow Occur(e, p_i, t') \wedge Within(t_a, l, u, t')$$

(h) *Validity axiom*: ..... (va)

$$Enable(e, t_a, t) \longrightarrow Trigger(f, t_a) \wedge Within(t_a, 0, u, t')$$

The axioms ensure sufficient completeness of the logical behavior of a TROM with respect to a set of signals, in the sense that, for each predicate of the form *Hold*(), *Occur*(), *Enable*(), *Disable*(), *Trigger* and for each state, event, and port, we can derive whether or not the predicates are true at each time instant. Moreover, only *Occur* predicate can be derived as a logical consequence. That is, only constrained and not input events can be derived from the axioms. Consequently, during the simulation process of a **TROM** when an event is scheduled at a port at a certain time, from the current state of the **TROM** the axioms will generate the successive state and the constrained events, if any. That is, all future states and events that are likely outcomes of a single step computation are derivable by the above axioms.

A subsystem is composed of a finite number of objects o_1,o_2,...,o_k. If the port p_j of object o_i is linked to the port q_r of the object o_k. then it is required that the occurrence of an event o_i.e at the port p_j should synchronize with the occurrence of the event o_k.e in the port q_r. This constraint is captured by the synchrony axiom

Synchrony axiom: ..... (SE)

For each link $o_i.@q_j \leftrightarrow o_k.@q_l$ in the SCS of the subsystem, a synchronous axiom is defined as follow:

$$\forall e \in \Sigma^p \bullet Occur(o_i.e, p_j, t) \longleftrightarrow Occur(o_k.e, q_l, t)$$

The **TROM** axioms and synchronization axioms taken together describe the logical behavior of a system. If a property, as stated in the requirement, is to be verified against the system design, then the property must be stated as a temporal formula using the above mentioned predicates and then must be proved to be a consequence of **TROM** and synchronization axioms. This is indeed one of the goals of the verifier within the animation tool.

The Axiom Generator module prepares the axioms for the specific **TROM** models constituting a system design. This is achieved by textually storing the **TROM** and synchrony axioms and then generating axioms for the **TROM**s in the system using the Abstract Syntax Trees, the internal **TROM** representations. The generated axioms are stored in a linked list for efficient retrieval and usage during the verification stages of the animator.

7.2 Axiom Generation

The architecture design for the Axiom Generator is shown in Figure 27.

There are two phases to generating the axioms. During the first phase the axioms stated in the previous section are read from a file, parsed using the grammar shown in Table 16 for syntactic correctness, and values for as many arguments as possible are substituted from the AST. Not every argument's value is known at this stage. So,

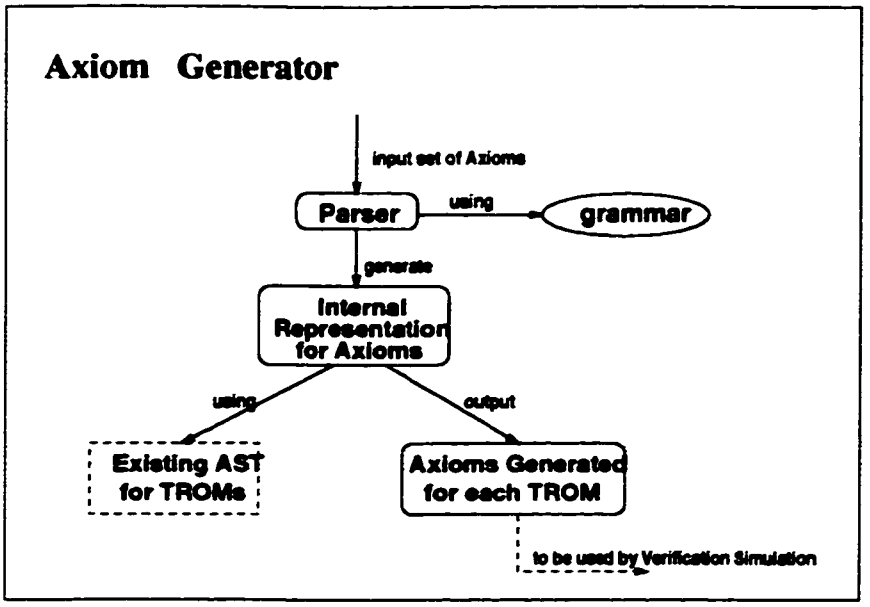


Figure 26: The Architecture for Axiom Generator

a second stage is necessary to instantiate the axioms and this is described in Section 4.

The states, events, transition specifications, the attributes, and the time constraints are known for a **TROM** from its specification and hence known from AST. However, the actual times of occurrences of events, the number of ports, and the ports at which events occur are not known from AST. Rather, we know them at the instance of instantiating object configurations.

The parsing phase is rather straightforward: an axiom is read, and parsed using the grammer in Table 16. Because we deal with a fixed number of predicate names, we have built a table driven parser(see Table 17). Based on the predicate name, the parser will know from the table the exact number and type of arguments to expect.

A token can be a predicate expression (not involving predefined predicate names) or a predicate name. In the former case, the grammer for a predicate expression as shown in Table 8(Chapter 4) will be used. Since every axiom is of the form LHS $\longrightarrow$ RHS, the parser ignores $\longrightarrow$ after a succesful parse and stores only LHS and RHS predicate formulas.

Axioms	::=	<*axiom_list*>
axiom_list	::=	<*axiom*> | <*axiom*>: <*axiom_list*>
axiom	::=	<*axiom_id*> : <*LHS*> $ <*RHS*>
LHS	::=	<*lterms*>
lterms	::=	<*lterm*> | <*lterm*> **AND** <*lterms*>
lterm	::=	<*predicates*>(<*arg_list*>) | <*arg*> <*predicates*> <*arg*> | <*arg*>(<*arg*>)
arg_list	::=	<*arg*> | <*arg*>, <*arg_list*>
predicates	::=	**Occur** | **Hold** | **Meet** | <> | **member** | = | **Not Occur Enable** | **Disable** | **Trigger** | **Within** | **Not member**
RHS	::=	<*rterm_list*> | <*rterm_list*> **OR** <*rterm_list*> | ϵ
rterm_list	::=	<*rterm*> | <*rterms*> **AND** <*rterm*>
rterm	::=	<*predicates*>(<*arg_list*>) | **NOT**<*predicates*>(<*arg_list*>) | <*arg*> <*predicates*> <*arg*> | <*enab_cond*> | <*port_cond*> | <*post_cond*> | {**forall** < *arg_list* >} | {**forall** < *arg* > $\in$ < *arg* >}
axiom_id	::=	CHAR(10)
arg	::=	CHAR(10)
enab_cond	::=	CHAR(50)
port_cond	::=	CHAR(50)
post_cond	::=	CHAR(50)

Table 16: A grammar for Axiom Generator

Predicate Name	Number of Arguments
Occur	3
Not Occur	3
Hold	2
Meet	2
Meet*	3
Enable	3
Disable	2
Trigger	2
Within	4
=	2
<>	2
belongto	2
Not belongto	2

Table 17: Predicates for Axiom Generator

For each **TROM** in the design, we generate axioms corresponding to the eleven **TROM** temporal axioms. The generation procedure is based on the following rules:

1. Axioms with *Occur* Predicate.

   For each event in a **TROM** specification the port type at which it occurs is known, but not the actual port is known at this time. Similarly, the time of occurrence is still not known. So, an event name from the **TROM** specification is substituted in *Occur* predicate, leaving the other two parameters as variables; however, from AST, the port type should be added on to the port argument. For example, the generation procedure for *Atomic-event axioms* of Train **TROM** is as follows:

```
for each event e ∈ ℰ do
    generate LHS axiom as
    Occur(e,p.t)
    for each event f ∈ ℰ do
        if e ≠ f then
        generate the RHS axiom
        Occur(f,q,t)
        endif
    endfor
endfor
```

   In the above algorithm, **p** is the port type at which **e** occurs and **q** is the port type at which **f** occurs. Note that this information is available in the AST corresponding to the **TROM**.

2. Axioms with *Hold* Predicate.

   For each state in a **TROM** specification the *Hold* and *HoldFor* predicates can be generated. Many of the axioms can be simplified due to value substitution. For

example, if θ_1 and θ_2 are two distinct simple states in the **TROM** specification, the *State-Uniqueness axiom* becomes

$Hold(\theta_1.T) \longrightarrow Hold(\theta_2.T)$

However, if θ_1 is a superstate of θ_2 in the TROM, the axiom will become

$Hold(\theta_2,T) \longrightarrow Hold(\theta_1,T)$

In fact, the above axiom will be written for each state θ_2 having θ_1 as a superstate. The AST for the **TROM** includes the information on the nature of state (simple or complex) and the substates, if admissible. Consequently, the generation algorithm is straightforward:

```
for each state θ1 ∈ S do
    generate LHS axiom as
    Hold(θ1,T)
    for each state θ2 ∈ S do
        if θ2 ≠ θ1 then
           if θ2 ∈ Φs(θ1) then
           generate RHS axiom as
           Hold(θ2,T)
           endif
           endif
    endfor
endfor
```

3. Tansition Axioms.

   For each transition specification, we generate one transition axiom. The dual of this axiom is the persistent axiom, and there is one such axiom corresponding to each transition axiom. These axioms involve both *Occur* and *Hold* predicates. The pre- and post- states corresponding to a state transition, and the transition specification causing the state transition give the names of states, and event for value substitution. For example, the algorithm that generates a Transition axiom follows:

```
for each transition_specification ts do
    generate LHS as the conjunction of the predicates
             Hold(θ,T), Occur(e,p,T), Meet(T,T'), where
    θ is the pre-state, and e is the event in ts
    generate the RHS as the conjunction of the predicates
             Hold(θ',T), post-condition in ts
endfor
```

4. Time Constraint Axioms.

The time constraint axiom consists of eight axioms and are the most important to assert the reactive behavior of a **TROM**. The arguments in the temporal predicates on intervals need no substitution. In all other predicates only the event name need to be substituted. Hence the axiom generation follows the rule for the *Occur* predicate. For example, the algorithm for generating the Constrained-event axiom follows:

```
for each time constraint v ∈ Γ do
    if e is the event in the transition specification of v
       and f is the constrained event in v then
       LHS is Occur(e,p,t)
       RHS is the conjunction of the predicates
       Occur(f,q,ta), Within(t_a,a,b,t)
       where [a,b] ∈ v and t_a is the activation instant of e
    endif
endfor
```

Notice that t_a will be known only at system execution stages and hence can not be substituted with any value at this stage. For those events **e** that are not constrained by any time-constraint there does not exist any activation instant t_a. That is, the above axioms exist only due to time-constrained specifications.

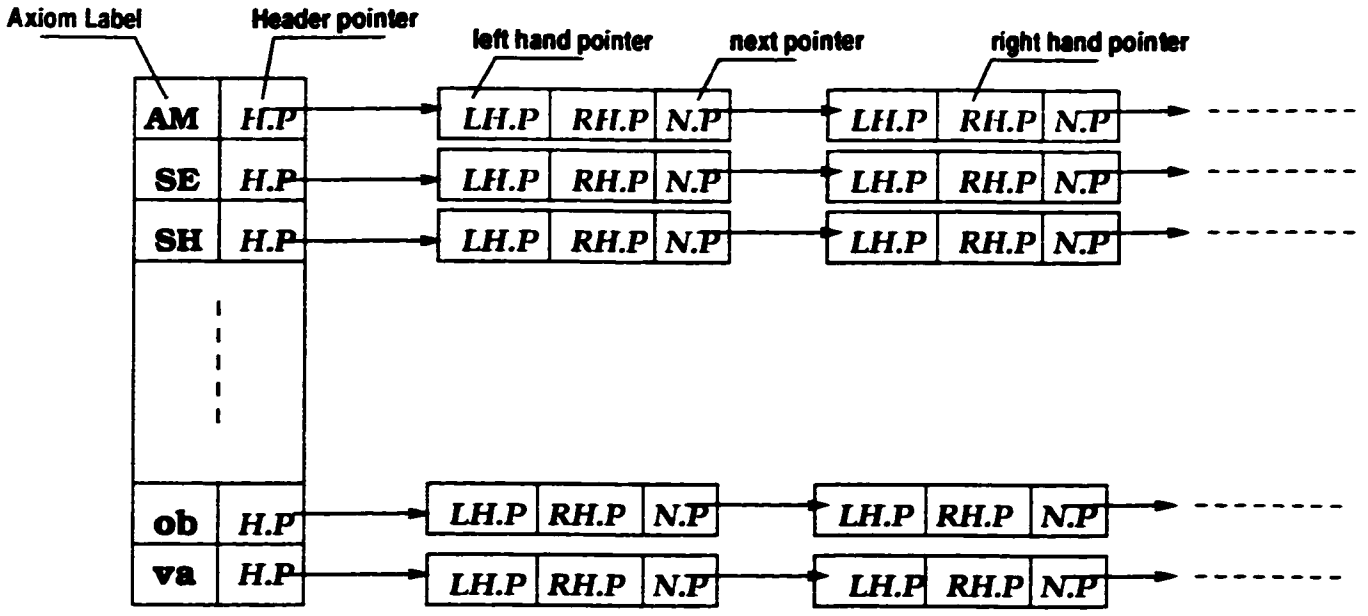


Figure 27: The Top Level Representation of **TROM** Axioms

7.3 Internal Representation for Axioms

Every axiom is of the form LHS $\longrightarrow$ RHS, where LHS and RHS are first order formulas. We represent this internally by a linked list with one header node describing the axiom name (eg; TR to denote Transition Axiom) and with a pointer to a node having three fields: LH_Pointer and RH_Pointer, which in turn reference the linked list representations for LHS and RHS respectively, and Next is the reference to the next axiom in this category. In fact, the header nodes of all axioms are modeled as a two-dimensional array structure of records, with first field labeled by Axiom_Label and the second field labeled by H_pointer. If the array name is Axioms and Axioms[1]. Axiom_Label is "AE" (denoting Atomic-event axiom), then Axioms[1].H_pointer will be a pointer to the header list representing the LHS and RHS of one Atomic-event axiom for a TROM. All other Atomic-event axioms of this TROM are linked through the Next field of this node. There will be one such structure for each TROM. See Figure 27 for this top level representation.

The LHS and RHS parts of an axiom are first order formulas involving named predicates and predicate expression. A predicate expression is represented by a binary tree, as has been explained previously in Chapter 5. An expression involving named

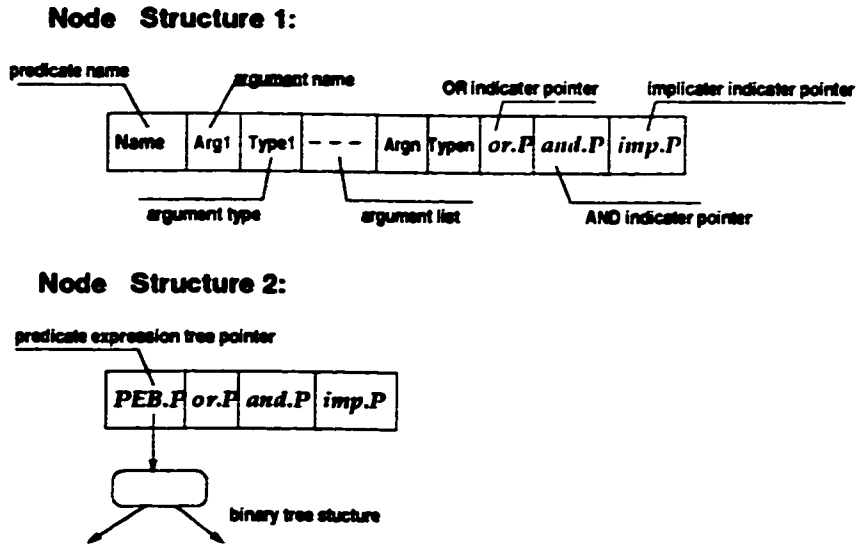


Figure 28: The LHS and RHS nodes structure

predicates is represented by a linked list of nodes, where each node has the structure shown in Figure 28.

The **name** field stores the predicate name (*Hold*, for example), the **arg_list** field stores the arguments of the predicate with their names and types, and the three fields **orp**, **andp**, **impp** are respective pointers to the nodes of named predicates or predicate expressions connected by *or*, *and*, *imp* to the predicate stored in this node. See Figure 29, which shows the representation for the *Transition* axiom for the Train TROM of the case study.

Axioms for a specific TROM are generated only when a request for it is received from the animator and the pointer to the structure is returned to the animator. Axioms that are no longer necessary for verification will be deleted by the animator. That is, the axiom generator only generates and represents axioms internally; their consumption and destruction is left for the animator.

7.4 How the axioms are used for verification

A full description of TROM based system verification is outside the scope of this thesis. In this section a brief account of the justification of the axiom generator design in the context of the verification is given.

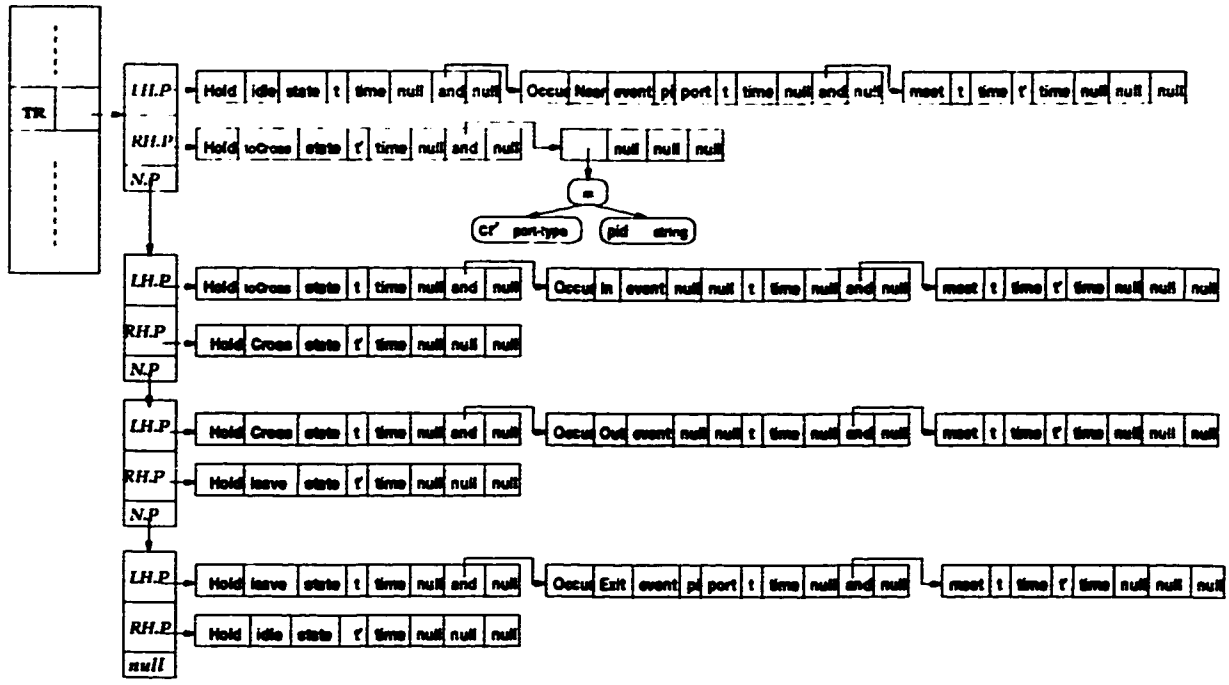


Figure 29: Transition Axiom for Train **TROM**.

Formal verification of real-time reactive systems is a complex task. Theoretically, such a process may not terminate, even for the verification of very simple properties. To reduce this complexity and at the same time deal effectively with non-trivial verification of system properties the verifier must be integrated with the simulator. During the simulation the values for timing parameters, the actual ports at which events occur and history of computation (which constitutes the knowledge of system activity) become known. This information when substituted for the parameters in the axioms that have been generated and stored makes the predicates as propositions, and inferring properties from a propositional system is guaranteed to terminate. It is towards this goal that we have implemented the axiom generator.

In order to prove properties, the verifier should generate proof tactics and identify the nature of axioms that might be useful. For example, to verify that an outcome is possible given the current configuration, one might try to use the time-constraint axioms, transition axioms and persistent axioms. Upon deciding, the verifier can invoke the axiom generator with the key *TC* (for Time-constraint axiom) and retrieve all axioms from the internal representation. Converting the axioms to propositions, and

conducting verification based on logical inference principle are part of the verification system.

When the system evolves due to the introduction of new **TROM**s, their axioms are generated from their ASTs incrementally.

7.5 Case Study – Axioms for Train-Gate-Controller Example

By using the algorithms given in section 2 and the AST of the Train, Gate and Controller **TROM**s, we obtain all axioms necessary for verification proterties. These specific axioms are enumerated below; see Figure 24 for the internal representation constructed during the derivation of transition axioms for Train **TROM**.

7.5.1 Axioms for Train TROM

1. *Atomic-event axiom*

$$
\begin{aligned}
Occur(Near,\ p_i,\ t) \quad &\longrightarrow \quad \neg\ Occur(Exit,\ P_j,\ t) \wedge \\
&\quad\ \neg\ Occur(In,\ NULL,\ t) \wedge \neg\ Occur(Out,\ NULL,\ t) \\
Occur(Exit,\ p_i,\ t) \quad &\longrightarrow \quad \neg\ Occur(Near,\ P_j,\ t) \wedge \\
&\quad\ \neg\ Occur(In,\ NULL,\ t) \wedge \neg\ Occur(Out,\ NULL,\ t) \\
Occur(In,\ NULL,\ t) \quad &\longrightarrow \quad \neg\ Occur(Near,\ P_j,\ t) \wedge \neg\ Occur(Exit,\ P_i,\ t) \wedge \\
&\quad\ \neg\ Occur(Out,\ NULL,\ t) \\
Occur(Out,\ NULL,\ t) \quad &\longrightarrow \quad \neg\ Occur(Near,\ P_j,\ t) \wedge \neg\ Occur(Exit,\ P_i,\ t) \wedge \\
&\quad\ \neg\ Occur(In,\ NULL,\ t)
\end{aligned}
$$

2. *Initial-state axiom*

   - $Hold(idle,\ t_{init})$

3. *Occurrence axiom*

$$Occur(Near, p_i, t) \longrightarrow Hold(idle, t)$$
$$Occur(In, NULL, t) \longrightarrow Hold(toCross, t)$$
$$Occur(Out, NULL, t) \longrightarrow Hold(cross, t)$$
$$Occur(Exit, p_i, t) \longrightarrow (Hold(leave, t) \wedge cr = pid)$$

4. *Transition axiom*

$$Hold(idle, t) \wedge Occur(Near, p_i, t) \wedge Meet(t, t') \longrightarrow Hold(toCross, t') \wedge cr' = pid$$
$$Hold(toCross, t) \wedge Occur(In, NULL, t) \wedge Meet(t, t') \longrightarrow Hold(cross, t')$$
$$Hold(cross, t) \wedge Occur(Out, NULL, t) \wedge Meet(t, t') \longrightarrow Hold(leave, t')$$
$$Hold(leave, t) \wedge Occur(Exit, p_i, t) \wedge Meet(t, t') \longrightarrow Hold(idle, t')$$

5. *Persistence axiom*

$$Hold(idle, t) \wedge Meet^*(T, t, t') \wedge \neg Occur(Near, p_i, t) \longrightarrow Hold(idle, t')$$
$$Hold(toCross, t) \wedge Meet^*(T, t, t') \wedge \neg Occur(In, NULL, t) \longrightarrow Hold(toCross, t') \wedge cr = cr'$$
$$Hold(cross, t) \wedge Meet^*(T, t, t') \wedge \neg Occur(Out, NULL, t) \longrightarrow Hold(cross, t')$$
$$Hold(leave, t) \wedge Meet^*(T, t, t') \wedge \neg Occur(Exit, p_i, t) \longrightarrow Hold(leave, t')$$

6. *Time-constraint axioms*

- Activation axiom:

$$Trigger(In, t_a) \wedge \neg Disable(In, t) \wedge Meet(t_a, t) \longrightarrow Enable(In, t_a, t)$$

$Trigger(Exit, t_a) \wedge \neg\, Disable(Exit, t) \wedge Meet(t_a, t)$
$\longrightarrow Enable(Exit, t_a, t)$

- Constraint-event axiom:

$Occur(In, NULL, t) \longrightarrow Occur(Near, p_i, t_a) \wedge Within(t_a, 2, 4, t)$

$Occur(Exit, p_i, t) \longrightarrow Occur(Near, p_i, t_a) \wedge Within(t_a, 0, 6, t)$

- Enabling axiom:

$Enable(In, t_a, t) \wedge \neg\, Occur(In, NULL, t) \wedge Meet(t, t') \wedge$
$\neg\, Disable(In, t') \longrightarrow Enable(In, t_a, t')$

$Enable(Exit, t_a, t) \wedge \neg\, Occur(Exit, p_i, t) \wedge Meet(t, t') \wedge$
$\neg\, Disable(Exit, t') \longrightarrow Enable(Exit, t_a, t')$

- Disabling axiom:

$Enable(In, t_a, t) \wedge Meet(t, t') \wedge Disable(In, t')$
$\longrightarrow \neg\, Enable(In, t_a, t')$

$Enable(Exit, t_a, t) \wedge Meet(t, t') \wedge Disable(Exit, t')$
$\longrightarrow \neg\, Enable(Exit, t_a, t')$

- Firing axiom:

$Enable(In, t_a, t) \wedge Occur(In, NULL, t) \wedge Within(t_a, 0, 4, t) \wedge$
$Meet(t, t') \longrightarrow \neg\, Enable(In, t_a, t')$

$Enable(Exit, t_a, t) \wedge Occur(Exit, p_i, t) \wedge Within(t_a, 0, 6, t) \wedge$
$Meet(t, t') \longrightarrow \neg\, Enable(Exit, t_a, t')$

- Prohibition axiom:

$Enable(In, t_a, t) \wedge Within(t_a, 0, 2, t) \longrightarrow \neg\, Occur(In, NULL, t)$

$Enable(Exit, t_a, t) \wedge Within(t_a, 0, 0, t) \longrightarrow \neg\, Occur(Exit, p_i, t)$

- Obligation axiom:

$Enable(In, t_a, t) \wedge \forall\, t'[\, Within(t_a, 0, 4, t') \longrightarrow \neg\, Disable(In, t')]$
$\longrightarrow Occur(In, NULL, t') \wedge Within(t_a, 2, 4, t')$

$Enable(Exit, t_a, t) \wedge \forall\, t'[\, Within(t_a, 0, 6, t') \longrightarrow \neg\, Disable(Exit, t')]$
$\longrightarrow Occur(Exit, p_i, t') \wedge Within(t_a, 0, 6, t')$

- Validity axiom:

$$Enable(In, t_a, t) \longrightarrow Trigger(Near, t_a) \wedge Within(t_a, 0.4, t')$$

$$Enable(Exit, t_a, t) \longrightarrow Trigger(Near, t_a) \wedge Within(t_a, 0.6, t')$$

7.5.2 Axioms for Gate TROM

1. *Atomic-event axiom*

$$\begin{aligned}
Occur(Lower, s_i, t) &\longrightarrow \neg\, Occur(Raise, S_j, t) \wedge \neg\, Occur(Up, NULL, t) \wedge \\
&\quad \neg\, Occur(Down, NULL, t) \\
Occur(Raise, s_i, t) &\longrightarrow \neg\, Occur(Lower, S_j, t) \wedge \neg\, Occur(Up, NULL, t) \\
&\quad \wedge \neg\, Occur(Down, NULL, t) \\
Occur(Down, NULL, t) &\longrightarrow \neg\, Occur(Lower, S_j, t) \wedge \neg\, Occur(Raise, S_i, t) \wedge \\
&\quad \neg\, Occur(Up, NULL, t) \\
Occur(Up, NULL, t) &\longrightarrow \neg\, Occur(Lower, S_j, t) \wedge \neg\, Occur(Raise, S_i, t) \wedge \\
&\quad \neg\, Occur(Down, NULL, t)
\end{aligned}$$

2. *Initial-state axiom*

   - $Hold(opened, t_{init})$

3. *Occurrence axiom*

$$\begin{aligned}
Occur(Lower, s_i, t) &\longrightarrow Hold(opened, t) \\
Occur(Down, NULL, t) &\longrightarrow Hold(toClose, t) \\
Occur(Raise, s_i, t) &\longrightarrow Hold(closed, t) \\
Occur(Up, NULL, t) &\longrightarrow Hold(toOpen, t)
\end{aligned}$$

4. *Transition axiom*

$$\begin{aligned}
Hold(opened, t) \wedge Occur(Lower, s_i, t) \wedge Meet(t, t') &\longrightarrow Hold(toClose, t') \\
Hold(toClose, t) \wedge Occur(Down, NULL, t) \wedge Meet(t, t') &\longrightarrow Hold(Closed, t') \\
Hold(closed, t) \wedge Occur(Raise, s_i, t) \wedge Meet(t, t') &\longrightarrow Hold(toOpen, t') \\
Hold(toOpen, t) \wedge Occur(Up, NULL, t) \wedge Meet(t, t') &\longrightarrow Hold(opened, t')
\end{aligned}$$

5. *Persistence axiom*

$$Hold(opened, t) \wedge Meet^*(T, t, t') \wedge \neg Occur(Lower, s_i, t) \longrightarrow Hold(opened, t')$$
$$Hold(toClose, t) \wedge Meet^*(T, t, t') \wedge \neg Occur(Down, NULL, t) \longrightarrow Hold(toClose, t')$$
$$Hold(closed, t) \wedge Meet^*(T, t, t') \wedge \neg Occur(Raise, s_i, t) \longrightarrow Hold(closed, t')$$
$$Hold(toOpen, t) \wedge Meet^*(T, t, t') \wedge \neg Occur(Up, NULL, t) \longrightarrow Hold(toOpen, t')$$

6. *Time-constraint axioms*

- Activation axiom:

$$Trigger(Down, t_a) \wedge \neg Disable(Down, t) \wedge Meet(t_a, t) \longrightarrow Enable(Down, t_a, t)$$

$$Trigger(Up, t_a) \wedge \neg Disable(Up, t) \wedge Meet(t_a, t) \longrightarrow Enable(Up, t_a, t)$$

- Constraint-event axiom:

$$Occur(Down, NULL, t) \longrightarrow Occur(Lower, s_i, t_a) \wedge Within(t_a, 0, 1, t)$$

$$Occur(Up, NULL, t) \longrightarrow Occur(Raise, s_i, t_a) \wedge Within(t_a, 1, 2, t)$$

- Enabling axiom:

$$Enable(Down, t_a, t) \wedge \neg Occur(Down, NULL, t) \wedge Meet(t, t') \wedge \neg Disable(Down, t') \longrightarrow Enable(Down, t_a, t')$$

$$Enable(Up, t_a, t) \wedge \neg Occur(Up, p_i, t) \wedge Meet(t, t') \wedge \neg Disable(Up, t') \longrightarrow Enable(Up, t_a, t')$$

- Disabling axiom:

$$Enable(Down, t_a, t) \wedge Meet(t, t') \wedge Disable(Down, t') \longrightarrow \neg Enable(Down, t_a, t')$$

$$Enable(Up, t_a, t) \wedge Meet(t, t') \wedge Disable(Up, t') \longrightarrow \neg Enable(Up, t_a, t')$$

- Firing axiom:

$$Enable(Down, t_a, t) \wedge Occur(Down, NULL, t) \wedge Within(t_a, 0, 1, t) \wedge Meet(t, t') \longrightarrow \neg Enable(Down, t_a, t')$$

$$Enable(Up, t_a, t) \wedge Occur(Up, p_i, t) \wedge Within(t_a, 0, 2, t) \wedge Meet(t, t') \longrightarrow \neg Enable(Up, t_a, t')$$

- Prohibition axiom:

  $Enable(Down, t_a, t) \wedge Within(t_a, 0, 0, t) \longrightarrow \neg\, Occur(Down, NULL, t)$

  $Enable(Up, t_a, t) \wedge Within(t_a, 0, 1, t) \longrightarrow \neg\, Occur(Up, NULL, t)$

- Obligation axiom:

  $Enable(Down, t_a, t) \wedge \forall\, t'[\, Within(t_a, 0, 1, t') \longrightarrow \neg\, Disable(Down, t')]$
  $\longrightarrow Occur(Down, NULL, t') \wedge Within(t_a, 0, 1, t')$

  $Enable(Up, t_a, t) \wedge \forall\, t'[\, Within(t_a, 0, 2, t') \longrightarrow \neg\, Disable(Up, t')]$
  $\longrightarrow Occur(Up, NULL, t') \wedge Within(t_a, 1, 2, t')$

- Validity axiom:

  $Enable(Down, t_a, t) \longrightarrow Trigger(Lower, t_a) \wedge Within(t_a, 0, 1, t')$

  $Enable(Up, t_a, t) \longrightarrow Trigger(Raise, t_a) \wedge Within(t_a, 0, 2, t')$

7.5.3 Axioms for Controller TROM

1. *Atomic-event axiom*

$$
\begin{aligned}
Occur(Near, q_i, t) \longrightarrow\ & \neg\, Occur(Exit, Q_j, t) \wedge \neg\, Occur(Lower, R_j, t) \wedge \\
& \neg\, Occur(Raise, R_j, t) \\
Occur(Exit, q_i, t) \longrightarrow\ & \neg\, Occur(Near, Q_j, t) \wedge \neg\, Occur(Lower, R_j, t) \wedge \\
& \neg\, Occur(Raise, R_j, t) \\
Occur(Lower, r_i, t) \longrightarrow\ & \neg\, Occur(Near, Q_j, t) \wedge \neg\, Occur(Raise, R_j, t) \wedge \\
& \neg\, Occur(Exit, Q_j, t) \\
Occur(Raise, r_i, t) \longrightarrow\ & \neg\, Occur(Near, Q_j, t) \wedge \neg\, Occur(Lower, R_j, t) \wedge \\
& \neg\, Occur(Exit, Q_j, t)
\end{aligned}
$$

2. *Initial-state axiom*

   - $Hold(idle, t_{init})$

3. *Occurrence axiom*

$$
\begin{aligned}
Occur(Near, q_i, t) &\longrightarrow (Hold(idle, t) \vee (Hold(active, t) \wedge \neg member(pid, inSet)) \vee (Hold(monitor, t) \wedge \neg member(pid, inSet)))\\
Occur(Lower, r_i, t) &\longrightarrow Hold(active, t)\\
Occur(Exit, q_i, t) &\longrightarrow ((Hold(monitor, t) \wedge size(inSet) > 1 \wedge member(pid, inSet)) \vee (Hold(monitor, t) \wedge size(inSet) = 1 \wedge member(pid, inSet)))\\
Occur(Raise, r_i, t) &\longrightarrow Hold(deactive, t)
\end{aligned}
$$

4. *Transition axiom*

$$
\begin{aligned}
&Hold(idle, t) \wedge Occur(Near, q_i, t) \wedge Meet(t, t') \longrightarrow Hold(active, t') \wedge inSet' = insert(pid, inSet)\\
&Hold(active, t) \wedge Occur(Near, q_i, t) \wedge Meet(t, t') \longrightarrow Hold(active, t') \wedge inSet' = insert(pid, inSet)\\
&Hold(monitor, t) \wedge Occur(Near, q_i, t) \wedge Meet(t, t') \longrightarrow Hold(monitor, t') \wedge inSet' = insert(pid, inSet)\\
&Hold(active, t) \wedge Occur(Lower, r_i, t) \wedge Meet(t, t') \longrightarrow Hold(monitor, t')\\
&Hold(monitor, t) \wedge Occur(Exit, q_i, t) \wedge Meet(t, t') \longrightarrow Hold(monitor, t') \wedge inSet' = delete(pid, inSet)\\
&Hold(monitor, t) \wedge Occur(Exit, q_i, t) \wedge Meet(t, t') \longrightarrow Hold(deactive, t') \wedge inSet' = delete(pid, inSet)\\
&Hold(deactive, t) \wedge Occur(Raise, r_i, t) \wedge Meet(t, t') \longrightarrow Hold(idle, t')
\end{aligned}
$$

5. *Persistence axiom*

$Hold(idle,\ t) \wedge Meet^{*}(T,\ t,\ t') \wedge \longrightarrow Hold(idle,\ t') \wedge inSet = inSet'$
$\neg\ Occur(Near,\ q_i,\ t)$
$Hold(active,\ t) \wedge Meet^{*}(T,\ t,\ t') \wedge \longrightarrow Hold(active,\ t') \wedge inSet = inSet'$
$\neg\ Occur(Lower,\ r_i,\ t)$
$Hold(monitor,\ t) \wedge Meet^{*}(T,\ t,\ t') \wedge \longrightarrow Hold(monitor,\ t') \wedge inSet = inSet'$
$\neg\ Occur(Exit,\ q_i,\ t)$
$Hold(deactive,\ t) \wedge Meet^{*}(T,\ t,\ t') \wedge \longrightarrow Hold(deactive,\ t') \wedge inSet = inSet'$
$\neg\ Occur(Raise,\ r_i,\ t)$

6. *Time-constraint axioms*

- Activation axiom:

$Trigger(Lower,\ t_a) \wedge \neg\ Disable(Lower,\ t) \wedge Meet(t_a,\ t)$
$\longrightarrow Enable(Lower,\ t_a,\ t)$

$Trigger(Raise,\ t_a) \wedge \neg\ Disable(Raise,\ t) \wedge Meet(t_a,\ t)$
$\longrightarrow Enable(Raise,\ t_a,\ t)$

- Constraint-event axiom:

$Occur(Lower,\ r_i,\ t) \longrightarrow Occur(Near,\ q_i,\ t_a) \wedge Within(t_a,\ 0,\ 1,\ t)$

$Occur(Raise,\ r_i,\ t) \longrightarrow Occur(Exit,\ q_i,\ t_a) \wedge Within(t_a,\ 0,\ 1,\ t)$

- Enabling axiom:

$Enable(Lower,\ t_a,\ t) \wedge \neg\ Occur(Lower,\ r_i,\ t) \wedge Meet(t,\ t') \wedge$
$\neg\ Disable(Lower,\ t') \longrightarrow Enable(Lower,\ t_a,\ t')$

$Enable(Raise,\ t_a,\ t) \wedge \neg\ Occur(Raise,\ r_i,\ t) \wedge Meet(t,\ t') \wedge$
$\neg\ Disable(Raise,\ t') \longrightarrow Enable(Raise,\ t_a,\ t')$

- Disabling axiom:

$Enable(Lower,\ t_a,\ t) \wedge Meet(t,\ t') \wedge Disable(Lower,\ t')$
$\longrightarrow \neg\ Enable(Lower,\ t_a,\ t')$

$Enable(Raise,\ t_a,\ t) \wedge Meet(t,\ t') \wedge Disable(Raise,\ t')$
$\longrightarrow \neg\ Enable(Raise,\ t_a,\ t')$

- Firing axiom:

$$Enable(Lower, t_a, t) \wedge Occur(Lower, r_i, t) \wedge Within(t_a, 0, 1, t) \wedge Meet(t, t') \longrightarrow \neg Enable(Lower, t_a, t')$$

$$Enable(Raise, t_a, t) \wedge Occur(Raise, r_i, t) \wedge Within(t_a, 0, 1, t) \wedge Meet(t, t') \longrightarrow \neg Enable(Raise, t_a, t')$$

- Prohibition axiom:

$$Enable(Lower, t_a, t) \wedge Within(t_a, 0, 0, t) \longrightarrow \neg Occur(Lower, r_i, t)$$

$$Enable(Raise, t_a, t) \wedge Within(t_a, 0, 0, t) \longrightarrow \neg Occur(Raise, r_i, t)$$

- Obligation axiom:

$$Enable(Lower, t_a, t) \wedge \forall t'[\, Within(t_a, 0, 1, t') \longrightarrow \neg Disable(Lower, t')] \longrightarrow Occur(Lower, r_i, t') \wedge Within(t_a, 0, 1, t')$$

$$Enable(Raise, t_a, t) \wedge \forall t'[\, Within(t_a, 0, 1, t') \longrightarrow \neg Disable(Raise, t')] \longrightarrow Occur(Raise, r_i, t') \wedge Within(t_a, 0, 1, t')$$

- Validity axiom:

$$Enable(Lower, t_a, t) \longrightarrow Trigger(Near, t_a) \wedge Within(t_a, 0, 1, t')$$

$$Enable(Raise, t_a, t) \longrightarrow Trigger(Exit, t_a) \wedge Within(t_a, 0, 1, t')$$

Chapter 8

Conclusion

This thesis is a contribution to the development of one component of a tool that provides the environment for creating, editing, combining and animating **TROM**s, the building blocks of real-time reactive systems. The interpreter developed in this thesis is the front end to the animator, which is being built by Dharmalingam Muthiayen. An overall user interface design is being planned by Jaya Konnankotil.

Formal approaches to software development of complex computer systems can be beneficial only when they are supported by tools. With this in mind our research originated to provide the tool support for the **TROM** based methodology [1]. The research reported in this thesis has led to the development of an interpreter, which includes a syntax checker, a semantic analyzer and axiom generator.

The grammar for **TROM** can be modified to accommodate parametrized event specifications or continuous time constraints. The syntactic and semantic analyzers for an inherited **TROM** need to compile the inherited **TROM** specification; that is, the compilation process in not incremental over the inheritance hierarchy. In particular, if a state refinement is done, the refined **TROM** must be recompiled. An useful future work would be to make the compilation process incremental, although it seems difficult to perform. During the debugging stages of the animator, a **TROM** definition may have to be changed. These require recompilation of redefiened **TROM**s.

The axiom generator can work incrementally over **TROM** inheritance hierarchies. When a new state is added to a **TROM**, this state, the transitions incident at this state, the assignment vector at this state, and the time constraints, if any, on the event labelling the transitions affecting the state can be dealt with separately to create the additional axioms.

When a new transition is added between two existing states, the additional axioms once again can be created with the added specifications, without changing the set of accumulated axioms. The axiom generator will be invoked by the animator during the verification stages.

One of the essential future research objective should be to link the interpreter with a more primitive front end module implementing OMT like tool, in which it would be easier for a less sophisticated user to state the requirements of an object and get a **TROM** specification generated automatically.

Bibliography

[1] Ramesh Achuthan. *A Formal Model for Object-Oriented Development of Real-Time Reactive Systems.* Doctoral thesis in the Department of Computer Science, Concordia University, Montreal, Canada, Sep. 1995.

[2] Vassilka Kirova & Willhelm Rossak. Representing Architectural Design: A Central Issue in the Development of Complex Systems. In *First IEEE International Conference on Engineering of Complex Compter Systems, Ft.Lauderdale, Florida.* Nov. 1995

[3] V.S.Alagar, P.Colagrosso, A.Loukas, S.Narayanan, A.Protopsaltou. "Formal Specifications for Effective Black Box Reuse - Final Report", Department of Computer Science, Concordia University, Montreal Canada, March 1996.

[4] A.Pnueli. Application of temporal logic to specification and verification of reactive systems: a survey of current trends. In W.P de.Roever J.W.de.Bakker and G.Rozenberg, editors, *Current trends in concurrency.* LNCS 224, Springer-Verlag, 1986.

[5] J.S. Ostroff. *Temporal logic for real-time systems.* Research studies press ltd., 1989.

[6] J.V. Guttag, J.J. Horning, and A. Modet. Revised report on the Larch shared language(version 2.3). Technical Report 58, Digital Equipment Corporation Systems Research Center, 1991.

[7] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reilly & Associates,Inc., 1995.

[8] Russel Winder. *Developing C++ Software.* John Wiley & Sons Ltd., 1991.

[9] Roger Sessions. *Class Construction in C and C++: Object-Oriented Programming Fundamentals* P T R Prentice-Hall, Inc., 1992.

[10] P.Colagrosso. "*Formal Specification of C++ Class Interface for Software Reuse*", Master of Computer Science, Department of Computer Science, Concordia University, Montreal Canada, 1993.

[11] J. F. Allen. Towards a general theory of action and time. *Artificial Inteligence(23)*, 1984.

[12] David Lorge Parnas. Speech for Fighting Complexity. Communications Research Laboratory, Software Engineering Research Group, Department of Electrical and Computer Engineering, McMaster University Hamilton, Ontario Canade L8S 4K1.

[13] Ronald Mak. *Writing Compilers and Interpreters.* John Wiley & Sons, Inc. 1991.

[14] Alfred V.Aho, Ravi Sethi, Jeffrey D.Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company. 1988.

[15] *Specification of Software Systems.* Course lecture, Department of Computer Science, Concordia University, Montreal Quebec Canade H3G 1M8. 1991.

[16] Haifeng Qian, Eduardo B. Fernandez, Jie Wu. A Combined Functional and Object-Oriented Approach. In *First IEEE International Conference on Engineering of Complex Compter Systems, Ft.Lauderdale, Florida.* Nov. 1995

[17] Y.Cheon, G.Leavens. "*A Quick Overview of Larch/C++.*", Journal of Object-Oriented Programming, 7(6):39-49, October 1994.

Appendix A

Syntax Grammar implementation using Flex and Bison

```
%{
   /* simple TROM lexer 001 */
#include <stdio.h>
#include <string.h>
#include "cont.h"
#include "lsldef.h"
#include "tromdef.h"
#include "scsdef.h"
#include "compdef.h"
#include "trom01.h"

int  lookupKeywords(btree<btname_t> *kwds, char *id);
btree<btname_t> *buildKeywords( );
extern int   lineno;
btree<btname_t> *keywords = buildKeywords( );

%}

iden    [a-zA-Z]+[a-zA-Z0-9]*
```

```
ws      [\t ]+
intnum  -?[0-9]+
nl      \n
%%

{ws}    ;
{nl}    {lineno++;}

"<->"   {return AGGREGATE;}
"->"    {return IMPLY;}
::      {return RELATE;}
"=>"    {return INVOKE;}

">=" |
"<=" |
"<>" |
"="     {yylval.id = strdup(yytext); return COMPARE_OP;}
">"     {return GT;}
"<"     {return LT;}
Attribute[ \t]*-[ \t]*function  {return ATT_FUNC;}
Transition[ \t]*-[ \t]*Spec     {return TRANS_SPECS;}
Time[ \t]*-[ \t]*Constraints  {return TIME_CONSTRAINTS;}
{iden}  { yylval.id = strdup(yytext); return lookupKeywords(keywords, yytext); }
{intnum}        {yylval.num = atoi(yytext); return NUM;}
.       {return yytext[0];}

%%

btree<btname_t> *buildKeywords( )
{
        btree<btname_t> *tree = new  btree<btname_t>;
```

```
        // Trom class keywords
        enter(tree, strdup("Class"), CLASS);
        enter(tree, strdup("end"), END);
        enter(tree, strdup("Events"), EVENTS);
        enter(tree, strdup("States"), STATES);
        enter(tree, strdup("Attributes"), ATTRIBUTES );
        enter(tree, strdup("Traits"), TRAITS);
        enter(tree, strdup("Create"), CREATE);
        enter(tree, strdup("Includes"), INCLUDES);
        enter(tree, strdup("Introduce"), INTRODUCES);
        enter(tree, strdup("SCS"), SCS_TOK);
        enter(tree, strdup("Include"), INCLUDE);
        enter(tree, strdup("Instantiate"), INSTANTIATE);
        enter(tree, strdup("Configure"), CONFIGURE);
        enter(tree, strdup("Trait"), TRAIT);
        enter(tree, strdup("OR"), OR);
        enter(tree, strdup("AND"), AND);
        enter(tree, strdup("NOT"), NOT);
        enter(tree, strdup("pid"), PID);
        enter(tree, strdup("true"), LOGIC);
        enter(tree, strdup("false"), LOGIC);
        enter(tree, strdup("Int"), INT_T);
        enter(tree, strdup("Bool"), BOOL_T);
        enter(tree, strdup("SEL"), SEL);
        btname_t *p = tree->getroot();
        return tree;
}
int  lookupKeywords(btree<btname_t> *kwds, char *id)
{
        btname_t *kwd;
        if(kwd=find( kwds, id))
```

```
        {
           return kwd->gettype();
        }
        return ID;
}

%{
   /* simple TROM parser 001 */

#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include "cont.h"
#include "lsldef.h"
#include "tromdef.h"
#include "compdef.h"
#include "scsdef.h"
#include "tromaux.h"

extern int yylex();
int yyerror(char *s);
extern int warning(char *, char *);
extern char *progname;
extern int   lineno;
extern list<TROM> *All_TROMs;
extern list<SCS> *All_SCSs;
extern list<LSL> *All_Traits;
extern list<SimEvent> *All_SimEvents;
```

```
extern fstream ferr; // Attach out file to Stream

%}
%union{
        char            *id;
        int             num;
        list<LSL>       *LSL_lst;
        list<lsl_func>  *lsl_func_lst;
        list<TROM>      *TROM_lst;
        list<event>     *event_lst;
        list<state>     *state_lst;
        list<attribute> *attribute_lst;
        list<trait>     *trait_lst;
        list<att_func>  *attfunc_lst;
        list<name_t>    *name_lst;
        btree<expr_elmt> *expr_tr;
        list<expr_elmt> *expr_el_lst;
        list<trans_spec> *trans_spec_lst;
        list<state_pair> *state_pair_lst;
        list<time_constraint>   *time_constraint_lst;
        list<SCS>       *SCS_lst;
        list<instance>  *instance_lst;
        list<port>      *port_lst;
        list<configure> *configure_lst;
        list<SimEvent>  *SE_lst;
        LSL             *LSL_ptr;
        lsl_func        *lsl_func_ptr;
        TROM            *TROM_ptr;
        event           *event_ptr;
        state           *state_ptr;
        attribute       *attribute_ptr;
```

```
        trait           *trait_ptr;
        att_func        *attfunc_ptr;
        expr_elmt       *expr_el_ptr;
        trans_spec      *trans_spec_ptr;
        state_pair      *state_pair_ptr;
        trigger_event   *trigger_event_ptr;
        time_constraint *time_constraint_ptr;
        SCS             *SCS_ptr;
        instance        *instance_ptr;
        port            *port_ptr;
        name_t          *name_ptr;
        configure       *configure_ptr;
        SimEvent        *SE_ptr;

}
%token TRAIT
%token CLASS END EVENTS STATES ATTRIBUTES
%token   TRAITS CREATE INVOKE
%token ATT_FUNC PID IMPLY AND NOT OR TRANS_SPECS
%token  TIME_CONSTRAINTS
%token LOGIC COMPARE_OP LT GT INT_T BOOL_T
%token SCS_TOK AGGREGATE INCLUDES INTRODUCES
%token  INSTANTIATE INCLUDE CONFIGURE RELATE
%token SEL
%token <id>  ID
%token <num>  NUM
%type <id> anId
%token <id>  LOGIC
%token <id>  COMPARE_OP
%type <LSL_lst>  trait_classes
%type <LSL_ptr>  trait_cls
```

```
%type <name_lst> lsl_args, lslf_params, arg_list
%type <name_lst> incl_traits, incl_ts
%type <name_ptr> arg
%type <lsl_func_lst> lslfuncs, lsl_fs
%type <lsl_func_ptr> lsl_f
%type <TROM_lst>  trom_classes
%type <TROM_ptr>  trom_class
%type <event_lst> evts, Yevents
%type <event_ptr> Yevent, internal_event,
input_event, output_event
%type <state_lst> sts, state_set, state_list
%type <state_ptr> Ystate
%type <name_lst>  Yport_ts
%type <name_ptr>  Yport_type, lslf_ret
%type <attribute_lst>   atts, Yatt_list
%type <attribute_ptr>   Yatt
%type <trait_lst>       trts, Ytrait_list
%type <trait_ptr>       Ytrait
%type <attfunc_lst>     attfs, attf_list
%type <attfunc_ptr>     attf
%type <name_lst>        att_name_list
%type <expr_tr>         expr_tr
%type <expr_el_ptr>     expr, simple_expr, term,
factor, lsl_term, func_arg
%type <expr_el_lst>     func_args
%type <id>              b_op
%type <trans_spec_ptr>  tr_spec, tr_spec_ini
%type <trans_spec_lst>  tr_spec_list, tr_specs
%type <state_pair_lst>  ini_stat, stat_prs
%type <state_pair_ptr>  stat_pr
%type <trigger_event_ptr> trig_ev
```

```
%type <time_constraint_ptr> constrt
%type <time_constraint_lst> constrts, time_constrts
%type <name_lst>        tc_sts
%type <SCS_lst>         scs_classes
%type <SCS_ptr>         scs_cls
%type <name_lst>        incls, incl_list
%type <name_ptr>        incl
%type <instance_lst>    insts, inst_list
%type <instance_ptr>    inst
%type <name_ptr>        aTrom
%type <port_lst>        port_list
%type <port_ptr>        port_card
%type <configure_ptr>   conf
%type <configure_lst>   conf_list, confs
%type <SE_lst>          sev_lst, sevs
%type <SE_ptr>          sev
%%

start:          trait_classes trom_classes scs_classes sev_lst
                {
                   All_Traits=$1;
                   All_TROMs = $2;
                   All_SCSs = $3;
                   All_SimEvents=$4;
                   ValidateAllLSLs( ferr, 0, All_Traits);
                   ValidateAllTROMs( ferr, 0, All_TROMs, All_Traits);
                   ValidateAllSCSs( ferr, 0, All_SCSs, All_TROMs);
                   ValidateAllSimEvents( ferr, 0, All_SimEvents,
                                                  All_TROMs, All_SCSs);
                };
sev_lst:        SEL ':' sevs END {$$=$3;}
```

```
                | /* empty */{$$=NULL;};
sevs:           sevs ';' sev  {$$=$1; $$->add($3);}
                | sev  {$$=new list<SimEvent>; $$->add($1);};
sev:            anId ',' anId ',' anId ',' NUM
                  {$$=new SimEvent($1, $3, $5, $7); };
trait_classes:  trait_classes trait_cls{$$=$1; $$->add($2);}
                | trait_cls{$$=new list<LSL>; $$->add($1);};
trait_cls:      TRAIT ':' anId '(' lsl_args ')' incl_ts lslfuncs END
                  {$$=new LSL($3); $$->setArgs($5);
 $$->setIncludes($7); $$->setLsl_funcs($8);}
                | /* empty */{$$=NULL;};
lsl_args:       lsl_args ',' anId {$$=$1; $$->add(new name_t($3));}
                | anId {$$=new list<name_t>; $$->add(new name_t($1));};
incl_ts:        INCLUDES ':' incl_traits{$$=$3;}
                | INCLUDES ':' {$$=NULL;};
incl_traits:    incl_traits ',' anId {$$=$1; $$->add(new name_t($3));}
                |  anId {$$=new list<name_t>; $$->add(new name_t($1));};
lslfuncs:       INTRODUCES ':' lsl_fs {$$=$3;};
lsl_fs:         lsl_fs ';' lsl_f {$$=$1; $$->add($3);}
                | lsl_f {$$=new list<lsl_func>; $$->add($1);};
lsl_f:          anId ':' lslf_params IMPLY lslf_ret
                   {$$=new lsl_func($1); $$->setArgs($3);
   $$->setRet_type($5);};
lslf_ret:       anId {$$=new name_t($1, LITERAL_TYPE);}
                | INT_T {$$=new name_t("Int", INT_TYPE);}
                | BOOL_T {$$=new name_t("Bool", BOOL_TYPE);};
lslf_params:    lslf_params ',' anId {$$=$1; $$->add(new name_t($3));}
                | anId {$$=new list<name_t>; $$->add(new name_t($1));}
                | /* empty */{$$=NULL;};

trom_classes:   trom_classes trom_class
```

```
                    {  $$=$1;
                        if (!findByName( (Tlist_ptr)$$,
     $2->getname())) $$->add( $2);
                    else ferr<<"Semantic Error:
       Duplicated TROM Class Name: "
<<$2->getname()<<endl; }
                | trom_class
                    { $$ = new list<TROM>; $$->add( $1);};

trom_class:     CLASS anId '[' Yport_ts ']' evts sts atts trts
 attfs tr_specs time_constrts END
                    {
                        $$= new TROM($2);
                        $$->setPort_types($4);
                        $$->setEvents($6);
                        $$->setStates($7);
                        if($8 != NULL) $$->setAttributes( $8);
                        if($9 != NULL) $$->setTraits( $9);
                        if($10 != NULL) $$->setAtt_funcs( $10);
                        if($11 != NULL) $$->setTrans_specs( $11);
                        if($12 !=NULL) $$->setTime_constraints( $12);
                    };

Yport_ts:       Yport_ts ',' Yport_type
                    {  $$=$1;
                    if(!findSame( (Tlist_ptr)$$, $3)) $$->add( $3);
                    else{ ferr<<"Duplicated port type: ";
                          ferr<<$3->getname()<<" at line no. "
<<lineno<<endl;} }
                | Yport_type
                    { $$ = new list<name_t>; $$->add( $1);};
```

```
Yport_type:     '@' anId   {$$=new name_t($2);};

evts:           EVENTS ':' Yevents    {$$=$3;};
Yevents:        Yevents ',' Yevent
                    {  $$=$1;
                    if(!findSame( (Tlist_ptr)$$, $3)) $$->add( $3);
                    else{ ferr<<"Duplicated event name: ";
                          ferr<<$3->getname()<<" at line no. "
<<lineno<<endl;} }
                | Yevent
                    { $$ = new list<event>; $$->add( $1);};
Yevent:         internal_event    {$$=$1}
                | input_event     {$$=$1}
                | output_event    {$$=$1};
internal_event: anId    {$$=new event($1, INTERNAL_EVENT)};
input_event:    anId '?' anId    {$$=new event($1, INPUT_EVENT, $3)};
output_event:   anId '!' anId    {$$=new event($1, OUTPUT_EVENT, $3)};
anId:           ID    {$$=$1};

sts:            STATES ':' state_set    {$$=$3};
state_set:      '*' Ystate ',' state_list   {  $$=$4;  $2->setInit(TRUE);
                    if(!findSame( (Tlist_ptr)$$, $2)) $$->insert( $2);
                    else{ ferr<<" Duplicated sate name: ";
                          ferr<<$2->getname()<<" at line no. "
<<lineno<<endl;}};
state_list:     state_list ',' Ystate
                    {  $$=$1;
                    if(!findSame( (Tlist_ptr)$$, $3)) $$->add( $3);
                    else{ ferr<<"Duplicated sate name: ";
                          ferr<<$3->getname()<<" at line no.
      <<lineno<<endl;} }
```

```
                | Ystate
                    { $$ = new list<state>; $$->add( $1);};

Ystate:         anId '(' state_set ')'
                    {$$=new state($1, COMPLEX_STATE);
$$->setSub_States($3);}
                | anId
                    {$$=new state($1)};

atts:           ATTRIBUTES ':' Yatt_list
                    {$$=$3}
                | /* empty*/
                    {$$=NULL;} ;
Yatt_list:      Yatt_list ';' Yatt
                    {  $$=$1;
                    if(!findSame( (Tlist_ptr)$$, $3)) $$->add( $3);
                    else{
                         ferr<<" Duplicated attribute name: ";
                         ferr<<$3->getname()<<" at line no. "
<<lineno<<endl;
                        }
                    }
                | Yatt
                    { $$ = new list<attribute>; $$->add( $1);};

Yatt:           anId ':' '@' anId
                    {$$=new attribute($1, $4, PORT_TYPE)}
                | anId ':' anId
                    {$$=new attribute($1, $3, TRAIT_TYPE)};

trts:           TRAITS ':' Ytrait_list
```

```
                    {$$=$3}
                | /* empty*/
                    {$$=NULL;} ;
Ytrait_list:    Ytrait_list ',' Ytrait
                    {  $$=$1;  $$->add( $3);}
                | Ytrait
                    { $$ = new list<trait>; $$->add( $1);};
Ytrait:         anId '[' arg_list ',' anId ']'
                    {$$=new trait($1);
 $3->add(new name_t($5, TRAIT_TYPE));
                        $$->setType_args($3);};
arg_list:       arg_list ',' arg
                    {  $$=$1;  $$->add( $3);}
                | arg
                    { $$ = new list<name_t>; $$->add( $1);};
arg:            anId
                    {$$=new name_t($1, TRAIT_TYPE);}
                | '@' anId
                    {$$=new name_t($2, PORT_TYPE);};


attfs:          ATT_FUNC ':' attf_list
                    {$$=$3}
                | /* empty*/
                    {$$=NULL;};
attf_list:      attf_list  attf
                    {  $$=$1;  $$->add( $2);}
                | attf
                    { $$ = new list<att_func>; $$->add( $1);};
attf:           anId  IMPLY '{' att_name_list '}' ';'
                    {$$=new att_func($1); $$->setAtt_names($4);}
```

```
                | anId IMPLY '{'  '}' ';'
                    {$$=new att_func($1);};
att_name_list:  att_name_list ',' anId
                    {  $$=$1;  $$->add( new name_t( $3));}
                | anId
                 { $$ = new list<name_t>; $$->add(new name_t( $1));};

tr_specs :      TRANS_SPECS ':' tr_spec_ini
tr_spec_list {$$=$4; $$->insert($3);}
                | TRANS_SPECS ':' tr_spec_list {$$=$3;};
tr_spec_ini:    anId ':' ini_stat CREATE '(' ')' ';'
                {$$=new trans_spec($1, TRUE);
                 $$->setState_pairs($3);};
ini_stat:        LT anId GT ';' {$$=new list<state_pair>;
                                 $$->add(new state_pair($2, NULL));};
tr_spec_list:   tr_spec {$$=new list<trans_spec>; $$->add( $1);}
                | tr_spec_list  tr_spec
                   {$$=$1;
                    if (!findByName( (Tlist_ptr)$$,
$2->getname())) $$->add( $2);
                    else{ ferr<<"Semantic Error:
 Duplicated Transition-spec: ";
                          ferr <<$2->getname()<<endl;}};
tr_spec:        anId ':' stat_prs  ';' trig_ev ';'
expr_tr INVOKE expr_tr ';'
                   {$$=new trans_spec($1, FALSE);
                    $$->setState_pairs($3);
                    $$->setTriggerEvent($5);
                    $$->setCondition($7, PRECONDITION);
                    $$->setCondition($9, POSTCONDITION);};
stat_prs:       stat_pr
```

```
                    { $$ = new list<state_pair>; $$->add($1);};
                  | stat_prs ',' stat_pr
                    {  $$=$1;  $$->add( $3);};
stat_pr:           LT anId ',' anId GT   {$$=new state_pair($2, $4);};
trig_ev:          anId '(' expr_tr ')' {$$=new trigger_event($1, $3);}
                  | anId {$$=new trigger_event($1);};

time_constrts:    TIME_CONSTRAINTS ':' constrts {$$=$3;}
                  | /* empty ---- for test only */ {$$=NULL;} ;

constrts:         constrt { $$=new list<time_constraint>; $$->add($1);}
                  | constrts ';' constrt {$$=$1;
                         if(!findSame((Tlist_ptr)$$, $3)) $$->add($3);
                         else{ ferr<<"Semantic Error:
 Duplicated time-constraint name: ";
                               ferr<<$3->getname()<<" at line no. "
<<lineno<<endl;}};

constrt:          anId ':' '(' anId ',' anId ',' '[' NUM
',' NUM ']' ',' '{' tc_sts '}' ')'
                         {$$=new time_constraint($1, $4, $6,$9,$11);
                          $$->setState_names($15);};

tc_sts:           anId {$$=new list<name_t>; $$->add(new name_t($1));}
                  | tc_sts ',' anId {$$=$1; $$->add(new name_t($3));}
                  | /* empty */ {$$=NULL;};

expr_tr:          expr   {$$=new btree<expr_elmt>; $$->setroot($1);};
expr:             simple_expr {$$=$1;}
                  | simple_expr b_op simple_expr
                     {$$=new expr_elmt($2, OPERATOR_TYPE, BOOL_TYPE);
```

```
                    $$->addleft($1); $$->addright($3); };
b_op:           COMPARE_OP {$$=$1;}
                | LT {$$="<"}
                | GT {$$=">"};
simple_expr:    term
                    {$$=$1;}
                | term OR term
                    {$$=new expr_elmt("OR", OPERATOR_TYPE, BOOL_TYPE);
                     $$->addleft($1);$$->addright($3); };

term:           factor
                    {$$=$1;}
                | factor AND factor
                    {$$=new expr_elmt("AND", OPERATOR_TYPE, BOOL_TYPE);
                     $$->addleft($1);$$->addright($3); };

factor:         PID         {$$=new expr_elmt("pid", PID_TYPE);}
                | anId      {$$=new expr_elmt($1, LITERAL_TYPE);}
                | anId '''  {$$=new expr_elmt
($1, LITERAL_TYPE_PRIME);}
                | LOGIC     {$$=new expr_elmt
($1, BOOL_TYPE, BOOL_TYPE);}
                | NUM       {char s[10]; sprintf(s, "%d", $1);
                  $$=new expr_elmt
(strdup(s), INT_TYPE, INT_TYPE, $1);}
                | lsl_term   {$$=$1;}
                | NOT factor {$$=new expr_elmt
("NOT", OPERATOR_TYPE, BOOL_TYPE);
                              $$->addleft($2);}
                | '(' expr ')'  {$$=$2;};
```

```
lsl_term:       anId '(' func_args ')'
                    {   $$=new expr_elmt($1, FUNC_TYPE);
                        $$->setFunc_args( $3);
                    };

func_args:      func_args ',' func_arg {$$=$1; $$->add($3);}
                | func_arg {$$=new list<expr_elmt>; $$->add($1);};

func_arg:       PID {$$=new expr_elmt("pid", PID_TYPE);}
                | anId {$$=new expr_elmt($1, LITERAL_TYPE);}
                | lsl_term{$$=$1;}
                | /*empty*/ {$$=NULL;};

scs_classes:    scs_classes scs_cls{$$=$1;
                if (!findByName( (Tlist_ptr)$$, $2->getname()))
{ $$->add( $2);
                 ferr<<"Added new SCS "<< $2->getname()<<endl;}
                else ferr<<"Semantic Error: Duplicated SCS Name: "
   <<$2->getname()<<endl;
    | scs_cls{$$=new list<SCS>; $$->add($1);};
scs_cls:         SCS_TOK ID incls insts confs END
{$$=new SCS($2); $$->setIncludes($3);
                        $$->setInstances($4); $$->setConfigures($5);
                        ferr <<"Compiler Info: SCS "<<$2 <<
" syntax check passed." << endl;
                         ferr <<"Compiler Error: SCS "<<$2 <<
" has semantic errors!" << endl;
                        else
                         ferr <<"Compiler Info:  SCS "<<$2 <<
" semantic check passed." << endl; */};
incls:          INCLUDE ':' incl_list{$$=$3;}
```

```
                | /* empty */ {$$=NULL;};
incl_list:      incl_list ';' incl{$$=$1; $$->add($3);}
                | incl{$$=new list<name_t>; $$->add($1);};

incl:           ID{$$=new name_t($1);};
insts:          INSTANTIATE ':' inst_list{$$=$3;}
                | /* empty */{$$=NULL;} ;
inst_list:       inst_list ';'  inst {$$=$1;
                   if(!findSame( (Tlist_ptr)$$, $3)) $$->add( $3);
                   else{ ferr<<"Duplicated Object name: ";
                         ferr<<$3->getname()<<" at line no.
"<<lineno<<endl;} }
                | inst {$$=new list<instance>; $$->add($1);};

inst:           anId RELATE aTrom '[' port_list ']'{$$=new instance($1);
                        $$->setTrom($3);
                        $$->setPorts($5);};
port_list:       port_list ',' port_card {$$=$1; $$->add($3);}
                | port_card{$$=new list<port>; $$->add($1);};
port_card:      '@' anId ':' NUM{$$=new port($2,$4);};
aTrom:          anId{$$=new name_t($1);};
confs:          CONFIGURE ':' conf_list{$$=$3;}
                | /* empty */{$$=NULL;}  ;
conf_list:      conf_list ';' conf {$$=$1; $$->add($3);}
                | conf {$$=new list<configure>; $$->add($1);};
conf:           anId '.' '@' anId AGGREGATE anId '.' '@'
anId {$$=new configure($1,$4,$6,$9);};

%%

int warning(char *s, char *t) /* print warning message */
```

```
{
        cerr << progname<< ": " << s;
        if (t) cerr << t;
        cerr << " line #" << lineno << endl;
}

int yyerror(char *s)
{
    cerr << s << endl;
}
```

Appendix B

Abstract Syntax Tree Definition

```
//cont.h   Containers definition header
#ifndef __CONT_H
#define __CONT_H

struct slink{
slink *next;
slink(){next=0;}
slink(slink *p){next=p;}
};

class btreenode: public slink
{
public:
btreenode *next;
btreenode *left;
btreenode *right;
btreenode(){next=left=right=0;}
addnext(btreenode *p){next=p;}
addleft(btreenode *p){left=p;}
btreenode *getleft(){return left;}
btreenode *getright(){return right;}
```

```
addright(btreenode *p){right=p;}
virtual bool operator=(btreenode &btn)=0;
virtual bool operator>(btreenode &btn)=0;
};

class slist_base{
slink *head;
slink *cursor;
int size;
public:
slist_base(){ cursor = head = NULL; size=0;}
~slist_base(){ head = NULL;}
int insert(slink *p);
int append(slink *p);
slink *get(){cursor=head; return head;}
int  getSize(){ return size;}
slink *next(){ cursor=cursor->next; return cursor;}
};

class btree_base{
btreenode *root;
btreenode *cursor;
int size;
public:
btree_base(){ cursor = root = NULL; size=0;}
~btree_base(){ root = NULL;}
int setroot(btreenode *p){ root = p;};
btreenode *getroot(){return root;}
int insert(btreenode *p);
int addnext(btreenode *p);
int addleft(btreenode *p);
```

```
int addright(btreenode *p);
btreenode *get(){cursor=root; return root;}
int  getSize(){ return size;}
btreenode *next()
          { cursor=cursor->next; return cursor;}
btreenode *left()
          { cursor=cursor->left; return cursor;}
btreenode *right()
          { cursor=cursor->right; return cursor;}
};

template <class T> class btree:private btree_base
{
public:
btree ():btree_base(){};
~btree();
void insert(T *pE){btree_base::insert(pE);}
void setroot(T *pE){btree_base::setroot(pE);}
T *getroot(){return (T *)btree_base::getroot();}
T* get(){ return (T *) btree_base::get();}
T *next(){ return (T *) btree_base::next();}
T *left(){ return (T *) btree_base::left();}
T *right(){ return (T *) btree_base::right();}
int getSize(){ return  btree_base::getSize();}
};

template <class T> class list:private slist_base
{
public:
list ():slist_base(){};
~list();
```

```
void insert(T *pE){slist_base::insert(pE);}
void add(T *pE){slist_base::append(pE);}
T* get(){ return (T *) slist_base::get();}
T *next(){ return (T *) slist_base::next();}
int getSize(){ return  slist_base::getSize();}
};

class name_t:public slink
{
char *name;
int  type;
public:
name_t(char *n=NULL){ name = n;}
name_t(name_t *nt=NULL){ if(!nt)name=NULL;
            else { name = strdup(nt->getname());
                   type=nt->gettype();}}
name_t(char *n=NULL, int t){ name = n; type = t;}
void setname(char *n=NULL){ name = n;}
char *getname(){ return name;}
void settype(int t){ type = t;}
int  gettype(){ return type;}
name_t *find(char *n)
               { if (strcmp(name, n)==0) return this;
                     else return NULL;}
~name_t(){if(name) delete name;}
};

typedef  list<name_t> *Tlist_ptr;

#endif//CompDef.h  Compiler definition header file
#ifndef __COMPDEF_H
```

```
#define __COMPDEF_H

#include <stream.h>
#include <iostream.h>
#include<string.h>
#include<malloc.h>
#include<bool.h>
#include"cont.h"

const NONE_TYPE = 0;
const BOOL_TYPE = 1;
const INT_TYPE = 2;
const REAL_TYPE = 3;
const PID_TYPE = 4;
const LITERAL_TYPE = 5;
const LITERAL_TYPE_PRIME = 6;
const OPERATOR_TYPE = 7;
const FUNC_TYPE = 8;

class btname_t : public btreenode
{
char *name;
int   type;

public:
btname_t(char *n=NULL):btreenode(){name = n;}
btname_t(char *n=NULL, int t):
                btreenode(){name = n; type=t;}
~btname_t(){if(name) delete name;}
void print(ostream &sout, int mode);
char *getname(){ return name;}
```

```
void setname(char *n)
            { if(name) delete name; name = n;}
int gettype(){ return type;}
void settype(int t){ type =t;}
virtual bool operator=(btreenode &btn);
virtual bool operator>(btreenode &btn);
int compare(char *n);
};

class  expr_elmt: public btname_t
{
int   value;
name_t  *value_t;
list<expr_elmt> *func_args;
public:
expr_elmt(char *n=NULL, int t = NONE_TYPE,
           int val_t = NONE_TYPE)
   :btname_t(n,t) {func_args=NULL;
            value_t = new name_t(NULL, val_t);}
expr_elmt(char *n=NULL, int t, int val_t, int val)
   :btname_t(n,t) {func_args=NULL;
            value_t = new name_t(NULL, val_t);}
~expr_elmt(){};
void print0(ostream &sout, int mode);
void print(ostream &sout, int mode);
int getvalue(){ return value;}
void setvalue(int val){ value =val;}
int getvalue_type(){ return value_t->gettype();}
void setvalue_type(int val_t){ value_t->settype(val_t);}
name_t *getvalue_t(){ return value_t;}
void setvalue_t(char *vn, int val_t)
```

```
                { value_t =new name_t(vn, val_t);}
void setvalue_t(name_t *v_t)
             { if( v_t) value_t =new name_t(v_t->getname(),
                    v_t->gettype());
                           else value_t=NULL;}
list<expr_elmt> *getFunc_args(){return func_args;};
void setFunc_args(list<expr_elmt> *fas){func_args=fas;};
bool validate(ostream &fserr, int mode);
bool checkOperator(ostream &fserr, int mode);
bool checkFunction(ostream &fserr, int mode);
};

btname_t *enter(btree<btname_t> *nametree, char *tbf, int t);
btname_t *find(btree<btname_t> *nametree, char *tbf);

#endif // __COMPDEF_H
//LSLDef.h  LSL_Trait definition header file
#ifndef __LSLDEF_H
#define __LSLDEF_H

#include <stream.h>
#include <iostream.h>
#include<string.h>
#include<malloc.h>
#include<bool.h>
#include"cont.h"

//---------------------------LSL_Trait definitions -----------------

class lsl_func:public name_t
{
```

```
list<name_t> *args;
name_t *ret_type;

public:
lsl_func(char *n):name_t(n){args=NULL;ret_type=NULL;};
~lsl_func(){};
void setArgs(list<name_t> *as){args=as;};
list<name_t> *getArgs(){return args;};
void setRet_type(char *rt){ret_type=new name_t(rt);};
void setRet_type(name_t *rt){ret_type=rt;};
name_t *getRet_type(){return ret_type;};
void print(ostream &sout, int mode);
};

class LSL:public name_t
{
list<name_t> *args;
list<name_t> *includes;
list<lsl_func> *lsl_funcs;
public:
LSL(char *n=NULL):name_t(n)
              {args=NULL;includes=NULL;lsl_funcs=NULL;};
void setIncludes(list<name_t> *inc){includes=inc ;};
list<name_t> *getIncludes(){return includes;};
void setArgs(list<name_t> *as){args=as ;};
list<name_t> *getArgs(){return args;};
void setLsl_funcs(list<lsl_func> *lfs)
              {lsl_funcs=lfs ;};
list<lsl_func> *getLsl_funcs(){return lsl_funcs;};
void print(ostream &sout, int mode);
~LSL(){;}
```

```
};

#endif // __LSLDEF_H
//TROMDef.h  TROM definition header file
#ifndef __TROMDEF_H
#define __TROMDEF_H

#include <stream.h>
#include <iostream.h>
#include<string.h>
#include<malloc.h>
#include<bool.h>
#include"cont.h"
#include"compdef.h"
#include"lsldef.h"

const INTERNAL_EVENT = 0;
const INPUT_EVENT =1;
const OUTPUT_EVENT =2;
const   SIMPLE_STATE = 0;
const   COMPLEX_STATE = 1;
const   PORT_TYPE = 10;
const   TRAIT_TYPE = 11;
const   PRECONDITION = 0;
const   POSTCONDITION = 1;

class event:public name_t
{
name_t *port_t;
public:
event(char *en, int et, char *ptn=NULL);
```

```
void setPort_type_name(char *ptn)
            { port_t->setname(ptn);}
char *getPort_type_name()
            { return port_t->getname();}
~event(){ ;}
};

class state:public name_t
{
bool  init;
list<state> *sub_states;
public:
state(char *n, int t=SIMPLE_STATE, bool isInit=FALSE);
void print(ostream &sout, int mode);
void setInit(bool isInit){ init = isInit;}
bool  getInit(){ return init;}
list<state> *getSub_States(){ return sub_states;}
void setSub_States(list<state> *sts){sub_states = sts;}
~state(){; /* sub_states */ }
};

class TROM;

class attribute:public name_t
{
name_t  *att_type;
public:
attribute(char *n, char *att_tn, int att_tt);
void print(ostream &sout, int mode);
void setAtt_type_t(int at_t){ att_type->settype(at_t);}
int  getAtt_type_t(){ return att_type->gettype();}
```

```
void setAtt_type(name_t *at_t){att_type=at_t;}
name_t *getAtt_type(){ return att_type;}
void setAtt_typename(char *n){ att_type->setname(n);}
char *getAtt_typename(){ return att_type->getname();}
        bool validate(ostream &fserr, int mode, TROM *trom);
~attribute(){;}
};

class att_func:public slink
{
name_t *state;
list<name_t> *attributes;
public:
att_func(char *sn=NULL){ state = new name_t(sn);
attributes=NULL;}
void print(ostream &sout, int mode);
void setState_name(char *sn=NULL){ state->setname(sn);}
char *getState_name(){ return state->getname();}
name_t *getstate(){ return state;}
void setAtt_names(list<name_t> *ans){ attributes = ans;}
list<name_t> *getAtt_names(){ return attributes;}
~att_func(){if(state) delete state;}
};

class trait:public name_t
{
list<name_t> *type_args;
public:
trait(char *n):name_t(n) {}
void print(ostream &sout, int mode);
void setType_args(list<name_t> *tas){ type_args = tas;}
```

```
list<name_t> *getType_args(){ return type_args;}
        bool validate(ostream &fserr, int mode, TROM *trom);
~trait(){;}
};

class state_pair:public slink
{
name_t *state_name[2];
public:
state_pair(char *n1, char *n2=NULL){
state_name[0]=new name_t(n1);
state_name[1]=new name_t(n2); }
void setname(char *n=NULL, int indx)
                {state_name[indx]->setname(n);}
char *getname(int indx)
                { return state_name[indx]->getname();}
~state_pair(){int i; for( i=0; i<2; i++)
if(state_name[i]) delete state_name[i];}
};

class trigger_event:public name_t
{
btree<expr_elmt> *condition;
public:
trigger_event(char *n, btree<expr_elmt>
                          *expr=NULL):name_t(n)
{condition=expr;}
void setCondition(btree<expr_elmt> *cond)
{ condition = cond;}
btree<expr_elmt> *getCondition()
{ return condition;}
```

```
};

class trans_spec:public name_t
{
bool  isInit; //if it is init; init has one state
      // and it has no trigger event
list<state_pair> *state_pairs;
trigger_event *trigger;
btree<expr_elmt> *conditions[2];
public:
trans_spec(char *n, bool bIni):name_t(n)
{   isInit =bIni; trigger = NULL;
    state_pairs = NULL;
    conditions[PRECONDITION] =
    conditions[POSTCONDITION] = NULL;
}
void print(ostream &sout, int mode);
void setState_pairs(list<state_pair> *sts)
{ state_pairs = sts;}
list<state_pair> *getState_pairs()
{ return state_pairs;}
btree<expr_elmt> *getCondition(int which)
{ return conditions[which];}
void setCondition(btree<expr_elmt> *pre_cs, int which)
{ conditions[which] = pre_cs;}
void setTriggerEvent( trigger_event *te)
{trigger=te;}
trigger_event *getTriggerEvent()
{return trigger;}
bool checkPostCond(ostream &fserr, int  mode,
btree<expr_elmt> *post_cond, TROM *trom);
```

```
bool validate(ostream &fserr, int mode, TROM *trom);
~trans_spec(){ }
};

class time_constraint:public name_t
{
int   min, max;
name_t *trans_name;
name_t *ev_name;
list<name_t> *state_names;
public:
time_constraint(char *n, char *tn,
                          char *en, int mi, int ma);
void print(ostream &sout, int mode);
void setTrans_name(char *n=NULL){trans_name->setname(n);}
char *getTrans_name(){return trans_name->getname();}
name_t *getTrans_spec(){return trans_name;}
void setEv_name(char *n=NULL){ev_name->setname(n);}
char *getEv_name(){return ev_name->getname();}
name_t *getEvent(){return ev_name;}
int   getmin(){ return min;}
int   getmax(){ return max;}
list<name_t> *getState_names(){ return state_names;}
void setState_names(list<name_t> *sts)
             {state_names = sts;}
~time_constraint(){delete trans_name, delete ev_name;}
};

class TROM:public name_t
{
list<name_t> *port_types;
```

```
list<event> *events;
list<state> *states;
list<attribute> *attributes;
list<trait> *traits;
list<att_func> *att_funcs;
list<lsl_func> *lsl_funcs;
list<trans_spec> *trans_specs;
list<time_constraint> *time_constraints;
public:
TROM(char *name);
~TROM();
void print(ostream &sout, int mode);
bool validate(ostream &fserr, int mode,
                          list<LSL> *lsl_lst);
char *getName(){return getname();}
list<name_t> *getPort_types(){ return port_types;}
void setPort_types(list<name_t> *pt){port_types = pt;}
list<event> *getEvents(){ return events;}
void setEvents(list<event> *ev){events = ev;}
list<state> *getStates(){ return states;}
void setStates(list<state> *sts){states = sts;}
list<attribute> *getAttributes(){ return attributes;}
void setAttributes(list<attribute> *atts){attributes = atts;}
list<trait> *getTraits(){ return traits;}
void setTraits(list<trait> *trts){traits = trts;}
list<att_func> *getAtt_funcs(){ return att_funcs;}
void setAtt_funcs(list<att_func> *afs){att_funcs = afs;}
list<lsl_func> *getLsl_funcs(){ return lsl_funcs;}
void setLsl_funcs(list<lsl_func> *lfs){lsl_funcs = lfs;}
list<trans_spec> *getTrans_specs(){ return trans_specs;}
void setTrans_specs(list<trans_spec> *tss){trans_specs = tss;}
```

```
void setTime_constraints( list<time_constraint> *tts)
                {time_constraints=tts;}
list<time_constraint> *getTime_constraints()
               {return time_constraints;}
};

#endif// tromaux.h

//Printing mode constants

const STD_OUTPUT = 0;
const VERBOSE = 1;


//----------------------Expression tree utilities----------------------
int  isOperatorOfBools(char *op);
int  isOperatorOfNoBools(char *op);
int     isOperatorBeBoth(char *op);
bool  isUnaryOperator(char *op);
void  printExprTree(btree<expr_elmt> *expr_tree,
          ostream &sout, int mod);
bool setLsl_funcsType(btree<expr_elmt> *expr_tree,
          list<LSL> *lsls, ostream &fserr, int mode);
bool validateTROMCondition(ostream &sout, int  mode,
       btree<expr_elmt> *cond, TROM *trom, bool isPostCond);
bool validateTROMExpr_elmt(ostream &fserr, int mode,
       expr_elmt *ee, TROM *trom, bool isPostCond);
bool checkTROMFunction(ostream &fserr, int mode,
       expr_elmt *ee, TROM *trom, bool isPostCond);

//----------------------General utilities-----------------------------
```

```
name_t *findByName( list<name_t> *alist, char *n);
name_t *findSame( list<name_t> *alist, name_t *n);
name_t *getNameByOrder( Tlist_ptr alist, int no);
int findNameLstOrder( Tlist_ptr alist, name_t *n);
bool getPort(char *port, char **port_t, int *port_no);

bool getFuncDesc(char *funcname, char *attname,
                 TROM *trom, list<LSL> *lsls);
trait *findTraitByAtt_Typename(list<trait> *trts,
                 char *type_n);

//----------------------TROM utilities---------------------------------
bool validateStates(list<state> *states);
attribute *findTraitInAtts(list<attribute> *atts, char *name) ;
state *findStateByName(list<state> *states, char *name);
event *findEventInPort(list<event> *evs,
                       char *evname, char *port_t);
name_t *getTROMAttType(TROM *trom, char *instname);
name_t *getLslTraitType(name_t *lsl_t, Tlist_ptr lslArgs,
                        Tlist_ptr traitArgs);
list<lsl_func> *createLslFuncLst(ostream &sout,
              int mode, TROM *trom, list<LSL> *lsl_lst);
lsl_func *findLslFuncInTrom( TROM *trom, lsl_func *lsl_f);
bool isLslFuncAndArgsMatch(lsl_func *f1, lsl_func *f2);

//----------------------SCS utilities----------------------------------
bool validateScsIncls(ostream &fserr,
                    list<SCS> *scs_list, SCS *pscs);

//---------------System-wide General Validation utilities-------------
bool ValidateAllLSLs( ostream &sout, int mode,
```

```
                        list<LSL> *lsl_lst);
bool ValidateAllTROMs( ostream &sout, int mode,
        list<TROM> *trom_lst, list<LSL> *lsl_lst);
bool ValidateAllSCSs( ostream &sout, int mode,
        list<SCS> *scs_lst, list<TROM> *trom_lst);
bool ValidateAllSimEvents( ostream &sout, int mode,
        list<SimEvent> *simev_lst,
        list<TROM> *trom_lst, list<SCS> *scs_lst);
void PrintAllLSLs( ostream &sout, int mode, list<LSL> *lsl_lst);
void PrintAllTROMs( ostream &sout, int mode, list<TROM> *trom_lst);
void PrintAllSCSs( ostream &sout, int mode, list<SCS> *scs_lst);
void PrintAllSimEvents( ostream &sout, int mode,
                        list<SimEvent> *simev_lst);
typedef struct _node{
void *data;
struct _node *next;} node_t, *node_ptr;

typedef struct _tromast{
char *class_name;
node_t *port_types;
node_t *events;} TROM_AST_t, *TROM_AST_ptr;

typedef struct _port_type{
char *name;} port_type_t, *port_type_ptr;

typedef struct _event{
char *name;
port_type_ptr port_type;
int  type;} event_t, *event_ptr;

//SCSDef.h  SCS & LSL_Trait definition header file
```

```
#ifndef __SCSDEF_H
#define __SCSDEF_H

#include <stream.h>
#include <iostream.h>
#include<string.h>
#include<malloc.h>
#include<bool.h>
#include"cont.h"
#include"compdef.h"
#include"tromdef.h"

const VALID = 0;
const INVALID =-1;
const NOTFOUND =-2;
//---------------------------SCS definitions ------------------------
class port:public name_t
{
int cardinality;
public:
port(char *n=NULL, int c):name_t(n){cardinality=c;};
void setCardinal(int c){cardinality=c;}
int getCardinal(){return cardinality;}
~port(){;}
};

class instance:public name_t
{
name_t *trom;
list<port> *ports;
public:
```

```
instance(char *objname=NULL):name_t
                (objname){trom=NULL; ports=NULL;};
void setTrom(name_t *tr){trom=tr;}
name_t *getTrom(){return trom;}
void setPorts(list<port> *pp){ports=pp;}
list<port> *getPorts(){return ports;}
void print(ostream &sout, int mode);
bool validate(ostream &fserr,
                      int mode, list<TROM> *troms);
~instance(){;}
};

class SCS;

class configure:public slink
{
name_t *objname[2];
name_t *portname[2];
int validateI(ostream &fserr, int mode, int indx,
                  char *port_t, int port_no, SCS *pscs,
                  list<name_t> *scs_lst);
public:
configure(char *on1, char *pn1, char *on2, char *pn2):slink()
  {objname[0]=new name_t(on1); objname[1]=new name_t(on2);
   portname[0]=new name_t(pn1); portname[1]=new name_t(pn2);};
void setObjectname(char *n=NULL, int indx)
                   {objname[indx]->setname(n);}
char *getObjectname(int indx)
                   { return objname[indx]->getname();}
void setPortname(char *n=NULL, int indx)
                   {portname[indx]->setname(n);}
```

```
char *getPortname(int indx)
                    { return portname[indx]->getname();}
void print(ostream &sout, int mode);
bool validate(ostream &fserr, int mode,
                      SCS *pscs, list<name_t> *scs_lst);
~configure(){;}
};

class SCS:public name_t
{
int mark;
list<name_t> *includes;
list<instance> *instances;
list<configure> *configures;
public:
SCS(char *n=NULL):name_t(n)
   {mark=0; includes=NULL;instances=NULL;
                configures=NULL;};
void setmark(int m){ mark=m;}
int getmark(){return mark;}
void setIncludes(list<name_t> *inc){includes=inc ;};
list<name_t> *getIncludes(){return includes;};
void setInstances(list<instance> *ins){instances=ins ;};
list<instance> *getInstances(){return instances;};
void setConfigures(list<configure> *confs)
                     {configures=confs ;};
list<configure> *getConfigures(){return configures;};
void print(ostream &sout, int mode);
bool validate(ostream &fserr, int mode,
           list<TROM> *troms, list<SCS> *scs_lst);
~SCS(){;}
```

```
};

//---------------------------SimEvent definitions ------------------
class SimEvent:public name_t
{
name_t *instname;
name_t *portname;
        int     time;

public:
SimEvent(char *n, char *instn, char *portn, int t):name_t(n)
          {instname=new name_t(instn); portname = new name_t(portn);
           time = t;};
~SimEvent(){};
void setInstname(char *instn){instname->setname(instn);};
char *getInstname(){return instname->getname();};
void setPortname(char *portn){portname->setname(portn);};
char *getPortname(){return portname->getname();};
        void setEv_time(int t){ time = t;}
        int  getEv_time(){ return time;}
        bool validate( ostream &sout, int mode,
           list<SCS> *scs_lst, list<TROM> *trom_lst);

void print(ostream &sout, int mode);
};

#endif
```