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

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

ANIMATION AND FORMAL VERIFICATION OF REAL-TIME REACTIVE SYSTEMS IN AN OBJECT-ORIENTED ENVIRONMENT

Darmalingum Muthiayen

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

October 1996

Abstract

Animation and Formal Verification
of Real-Time Reactive Systems
in an Object-Oriented Environment

Darmalingum Muthiayen

Real-time reactive systems are characterized by their continuous interaction with their environment through stimulus-response behavior. The safety-critical nature of their domain and their inherent complexity advocate the use of formal methods in the software development process. TROMLAB development environment supports a process model adequate for dealing with the complexity of reactive systems. The foundation of the TROMLAB environment is the Timed Reactive Object Model (TROM), which combines object-oriented and real-time technologies.

Simulation is essential in the behavioral analysis of real-time reactive systems; animation allows a visualization of the simulation process. A rigorous trace analysis of simulation scenarios provides insight into the behavior of the collaborating entities in the configuration. This supports validation of systems designed incrementally and iteratively in the software development life-cycle. Moreover, safety-critical systems need to be verified for adherence to stringent safety and liveness properties.

The scope of this thesis is two-fold. We first present an animation tool supporting simulation of reactive systems described in the TROM formalism. We include formal specifications of the functionalities of the simulator in VDM specification language. We then introduce a methodology for formal verification of TROM subsystems. The novelty of the methodology lies in the formal verification approach embedded within an object-oriented framework. The simulator and the verification methodology conform respectively to the operational and logical semantics of TROMs.

To my parents.

Acknowledgments

I am forever indebted to my supervisor, Professor V. S. Alagar, for his continuous support, both technical and financial, throughout my studies. Dr. Alagar's technical assistance has been fundamental to the success of this work. His sagacity and discernment have indued the regular research meetings with the requisites for providing a rigorous framework conducive to progress. Besides, his uttermost kindness and conveyance of ethical codes have prompted the highest esteem from me. Dr. Alagar reviewed a preliminary version of this thesis and provided useful comments.

I gratefully acknowledge the International Fee Remission Award provided by the Government of Québec, the Concordia University Graduate Fellowship awarded to me by Concordia University, and the Graduate Partial Tuition Scholarship also awarded by Concordia University.

I am grateful to Dr. Alagar for giving me access to the equipment of the Softeks Lab. The system analysts of the Computer Science Department have provided effective technical support: I am appreciative to them. I thank the secretaries of the department for their courteous collaboration. I convey my gratitude to Claudette Fortier and Pat Hardt of the International Student Office for their continued assistance.

Ramesh Achuthan, the author of the TROM model, deserves my appreciation for his timely and inspiring discussions. Ramesh has always found time to answer my questions, and he was supportive in the development of the tool. I am grateful to the other members of the TROMLAB research group, Angela Tao, Jaya Konnankottil, and Rajee Nagarajan for their earnest friendship, and for the thought-provoking discussions. I am thankful to Antonis Protopsaltis and Sridhar Narayanan with whom I prototyped an interface for the TROM specification environment.

Several persons have somehow contributed to the success of this work; I am forever grateful to them. Dr. Kasi Periyasamy has continuously expressed encouragement, and Dr. T. Radhakrishnan has conveyed renewed commendation. Conchita and Sunil Beeharry-Panray have always lent an attentive ear. Corinne and Paul Fieldhouse, Sandra and Sanjaye

Ramdoyal, and Veena and Benode Dookun deserve my gratitude for their warmth and friendship. The graduate students who contribute to make the tenth floor of the McConnell Building a livable abode are assured of my gratitude.

I acknowledge the precious contribution of my parents, and my sisters and their families, who despite being on the other side of the globe, bring significance to the accomplishment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time reactive systems are increasingly used in control systems in a wide array of domains including telecommunications, air traffic control, nuclear reactors, robotics, and medicine. The safety-critical nature of the application domain and the intrinsic complexity of such systems call for a formal development environment supporting validation and verification. TROMLAB *is an environment which is being built to support a software development process model adequate for reactive systems. The contribution of this research is in the form of an animation tool supporting validation and a methodology for formal verification of real-time reactive systems.*

1.1 Real-Time Reactive Systems

The distinctive feature of a reactive system is the continuous interaction between the system and its environment. The system receives and sends messages through a hardware interface consisting of sensors and actuators, giving rise to stimulus-response behavior. The sequence of interactions depends on several factors, the most influential being the level of coupling between the entities in the environment. In the case of *real-time* reactive systems, stimulus-response behavior is regulated by timing constraints. Nuclear reactors and air traffic control systems are typical examples of *safety-critical* systems involving concurrency and synchronous communication between actuators, reactors and reactive entities. Common to all these applications is the notion of *reactive behavior*, wherein the relationship between input and output over time, complex sequencing of events and the way they constrain the computations are described.

Several factors contribute to the complexity of a real-time reactive system. These include largeness, criticality, concurrency, and the time-dependent nature of the real-world processes they control. The requirements of large reactive systems are generally difficult to comprehend, maintain, and modify. The complexity due to requirements permeates into several layers of detail, and if not properly understood and tackled will lead to faulty design causing very unpredictable and catastrophic system failures. This type of complexity cannot be avoided and should be resolved. Another type of complexity arises due to modeling, and software design. The language of specification and design must be clear, easy to learn and use, precise and formal, amenable to express changes and to inspect the consequences of actions and interactions. In the design, the level of coupling among system components must be low and the length of the shortest description, which indicates the amount of information required to understand the product, must be small. In other words, to quote Parnas [Par95], a system can be considered complex if its shortest useful description is relatively long. Hence, the purpose of design and documentation must be to minimize this complexity, especially when faced with complex requirements.

Interactions among system and environmental entities in a reactive system can be quite complex to describe; an entity can interact with several other entities during a certain interval of time to evoke a time-constrained behavior at a future time interval, if certain environmental conditions are met. That is, such interactions exhibit *non-determinism* in *time*, *control*, and *interaction*. During the development stages such requirements must be understood by the development team, who in turn should ensure the presence of these properties in the system and demonstrate it to users. Consequently, the specification method should support all activities in a life-cycle model - requirements to specification, design,

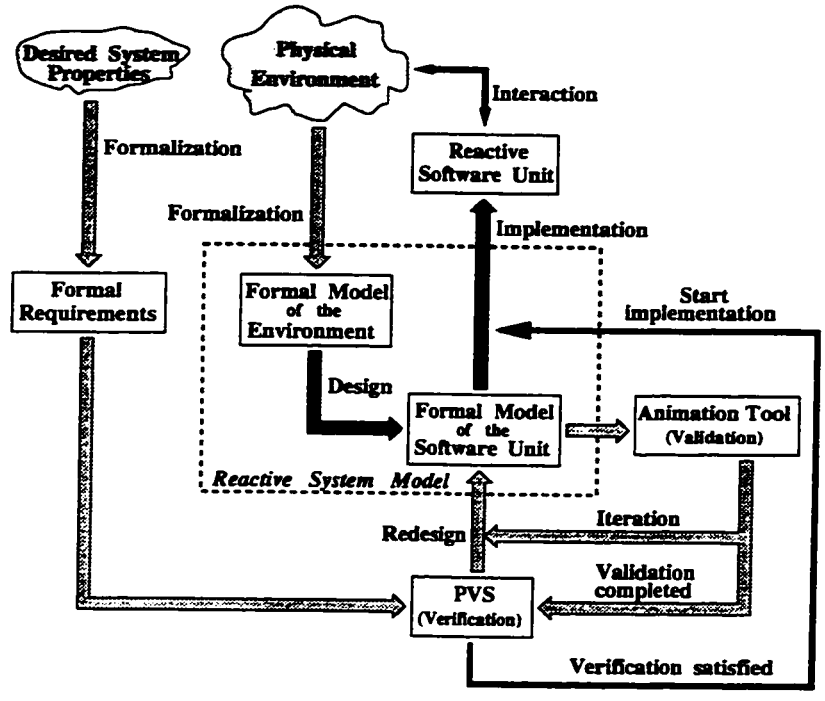


Figure 1: Process model for developing complex reactive systems.

prototyping, validation, and verification. Since a real-time reactive system for safety-critical applications, such as avionics, require on-line verification and validation, the specification language should be such that the design can be subjected to a formal analysis at any stage. That is, debugging, simulation, and verification should be made possible within the development framework.

1.2 Research Goals

Our goal is to build a software development environment supporting modeling, design, validation, and verification of real-time reactive systems. The complexity of reactive systems advocates the use of a formal framework with tool support for the design and experimentation processes. The overall objective of this research is to aid the development of real-time reactive systems supported by a formal object-oriented approach. This thesis makes two significant contributions towards achieving the above goals: (1) animation tool; and (2) verification methodology. The process model shown in Figure 1 aids an incremental and iterative development process. The animator, the brain child behind this process, is also the catalyst in reducing design complexity.

We give an overall view of the research with a brief description of TROMLAB development environment in Section 1.2.1. In Section 1.2.2, we emphasize the contribution of this

thesis with an outline of animation and formal verification of real-time reactive systems described in *Timed Reactive Object Model* (TROM) [Ach95].

1.2.1 TROMLAB Software Development Environment

TROMLAB is an object-oriented software development environment for reactive systems, supporting the process model shown in Figure 1. The model was introduced by Achuthan [Ach95]; it has been adapted to incorporate mechanized formal verification. The process model incorporates an *iterative development* approach, the benefits of which are well-known for:

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

In this process model, we first formalize the requirements and the environment model of the system, then design a formal model of the software, and use iterative development to validate the system, verify requirements and integrate the components. The model supports the development of complex real-time reactive systems described using TROM.

The environment provides a two-pronged strategy to contain complexity: the object-oriented framework for modeling reactive systems supports iterative system design and minimizes design complexity; the animator and the verification system provide the tool support necessary for validating design against requirements and verifying time-dependent properties during the evolution of design. The TROM formalism has a three-tiered structure which supports modeling real-time reactive systems using object-oriented techniques. It allows encapsulation of characteristics inherent to reactive systems in class descriptions, inclusion of abstract data types in the model described, and incremental configuration of reactive systems using objects, subsystems, and links. The specification environment of TROMLAB allows users to develop syntactically and semantically correct TROM classes, models describing the behavior of reactive objects. The design environment supports an incremental development of systems built on TROM objects and other subsystems. The facilities provided support design debugging, simulating a computational step and analyzing its consequences, and verifying invariant properties of an evolving design at different stages of the design process.

1.2.2 Animation and Formal Verification

An important goal of animating the simulation process is to facilitate design-time debugging, and validating design against requirements. Simulation allows observing the behavior of a system through a trace analysis of the simulated scenarios. Animation allows visualization of interactions among entities in the system under development, through a graphical user interface. The configuration of formally specified subsystems are validated, and timing constraints and properties are verified during the simulation process. Trace analysis of simulation scenarios provide invaluable insight into the behavior of the objects in the configuration, the subsystems incorporated, and the reactive system as a whole.

A simulation model should be capable of detecting faults in the design of a system. Such a model introduces *predictability* for properties that have to be maintained in the future. Sufficient information is required in the validation tool to verify properties and deduce reasons for a specific behavior. The *history* of event traces allows the user to roll the simulation clock backward to detect and fix faults in the design. The simulation process is also capable of predicting the behavior in order to analyze the properties that have to be maintained in the future. Consequences of refinements, changes to event occurrences, and time constraints can be analyzed before changes to the design are agreed upon. Incorporating a reasoning system in the simulation environment allows the use of deduction to verify properties of the system under development, based on the history of *computational steps*. Thus both validation and verification facilities are integrated in one toolset. Consequently, it becomes much easier to understand the behavior of the system under development during design evolution. The development environment provides facilities for modular design of TROM classes, modular composition of objects to build subsystems, and analysis capabilities which combine simulation and verification.

Reactive systems in safety-critical contexts need to adhere to stringent safety and liveness properties. A safety property entails that something wrong will never happen, while a liveness property implies that something good will eventually happen. Formal verification of such properties provides the assurance required for critical systems. A methodology for formal verification of TROM-based systems needs to conform to the logical semantics of the TROM model. The methodology proposed allows verifying properties in a systematic manner. Axioms are derived from the class specifications and subsystem configurations, and the property to be verified is expressed as an invariant assertion on the state of the system. The underlying object-oriented nature of the model allows for the verification process to be carried out in a modular fashion, thus dealing with the inherent complexity of reactive systems. The methodology can be related to the higher-order logic of PVS [ORS92].

1.3 Thesis Outline

Chapter 2 describes the TROMLAB software development environment. It briefly sketches the stages in the process model supported by TROMLAB. It then outlines the TROM formalism, including the computational model based on its operational semantics. A generalized version of the *Train-Gate-Controller* system is used as a case study to illustrate the formalism and the operational semantics. Chapter 2 also includes a brief description of the animation and verification features of TROMLAB. It concludes with a review of related works. Chapter 3 discusses the functionalities of the animation environment, including simulation and validation. It then describes the design of the animation tool, including its architecture and components. It also includes an outline of the components designed to support TROM. Chapter 4 presents a formal specification of the simulator in VDM specification language. It also shows how this exercise helped to validate the simulation algorithm so as to faithfully conform to the operational semantics of TROM. Chapter 5 gives an introduction to formal verification of real-time systems, including a description of mechanized verification. It includes an outline of the semantics of the since operator, and its relation to absolute times. It then introduces a methodology for formal verification of TROM-based systems. It also explains the steps leading from TROM axioms to proving safety properties. Chapter 6 illustrates the verification methodology using the Train-Gate-Controller example. It includes a detailed description of the proof steps. Chapter 7 concludes the thesis with an outline of its contribution and the future goals.

Chapter 2

TROMLAB **Environment**

The development environment provides facilities for modular design of TROM *classes, modular composition of objects to build subsystems, and analysis of system behavior, combining simulation and verification. Simulation helps to debug the design and validate design against requirements. Consequences of refinements, changes to event occurrences, and time constraints can be analyzed before changes to the design are brought out.*

2.1 Introduction

This chapter gives a detailed description of the TROMLAB software development environment. Section 2.2 outlines the stages in the process model supported by TROMLAB. Section 2.3 gives an informal description of the TROM formalism, including its syntax. It also outlines the operational semantics on which the simulator is built, and the logical semantics on which the verification methodology is based. As a case study, we illustrate the model using a generalized version of the Train-Gate-Controller system. Section 2.4 briefly sketches the animation tool and Section 2.5 introduces the verification methodology. Section 2.6 concludes the chapter with the related work.

2.2 Process Model

The process model shown in Figure 1 defines the series of software engineering stages that we follow to the development of real-time reactive systems. Iterative development, incremental design, and application of formalism through the different stages of development are the virtues of this process model. Following the paths in Figure 1, one can understand the orderly progress of development activity. The first step is to identify and formalize the desired properties of the physical environment, the context in which the final system is to operate. A formal environmental model is constructed by further abstracting these properties. Following this, a formal model of the software unit controlling the reactive system is designed. This stage involves identifying functional and timing requirements and producing their formal descriptions.

The desired properties of a reactive system are usually not expressible as the behavior of the software unit alone, instead they are statements about the cooperation between the software unit and the environment. Hence, to guarantee acceptable behavior of the software unit, a set of environmental behavior on which the software unit can rely has to be given. Therefore, a formal model of a reactive system is composed of a model of the software unit and a model of the environment in which it is embedded. Such models are called *closed system* models, since they are completely self-contained [Lam91]. In contrast, *open system* models do not define the behavior of the environment.

The benefits of formal methods in reactive system development are many-fold. The constructs in a formal specification language have well defined meanings. Any term other than the ones provided in a formal specification language is required to be defined by the specifier. Formal specifications can be subjected to formal deductions. Due to these reasons, imprecision, ambiguities, and inconsistencies in the requirements can be removed. Within

a formal framework, it is possible to conduct a rigorous analysis of software requirements for detecting safety-related software errors in embedded systems before their deployment. Formal specifications of component descriptions, interface descriptions, time dependent controls, and protocols for object collaborations break the complexity barrier in system design, and enable rigorous system reviews through validation, and verification.

The formal model of the reactive unit is not implemented until several iterations of design, as depicted in Figure 1, take place. The cycle involves the three stages: design validation, redesign (if necessary), and formal verification. Validation is done through *system simulation*, the central piece of an animation tool in the process model. Simulation uses only the formal model to generate observable behaviors, and hence is independent from any implementation decisions. Consequently, the behaviors can be directly related to requirements for their satisfaction. If flaws due to incorrect functionalities and/or inconsistent timed behavior are noticed during system simulation, the process model allows redefining the formal model of the reactive unit. After redefinition and redesign, the system is simulated again for validating requirements. This iterative process continues until only acceptable behaviors are observed in the formal model of the software unit.

System verification takes place at the next stage of the process model. This stage just precedes the implementation of the system. The desired system properties are formalized into two important kinds: *safety properties*, and *timeliness properties*. Formal verification of these properties is done using a methodology related to PVS [ORS92].

2.3 TROM Formalism

The three-tier structure of the object-oriented methodology introduced by Achuthan [Ach95] is shown in Figure 2 [Ach95]. The formalism is sufficiently expressive for modeling reactive systems. The benefits derived from the object-oriented techniques include *modularity* and *reuse*, *encapsulation*, and *hierarchical decomposition* using *inheritance*. Encapsulation in reactive systems is meaningful in associating attributes, properties, logical assertions, and timing constraints with specific classes of entities. Large and complex systems can be developed incrementally by composing, verifying, and integrating subsystems.

The three tiers independently specify system configurations, reactive objects, and abstract data types, by importing lower-tier specifications into upper tiers. TROM is a hierarchical finite state machine augmented with ports, attributes, logical assertions on the attributes, and time constraints. The middle-tier formalism specifies reactive objects as TROM classes. Abstract data types are specified as LSL (Larch Shared Language) [GH93] traits in the lowest tier, and can be used by objects modeled by TROM. The upper-most tier

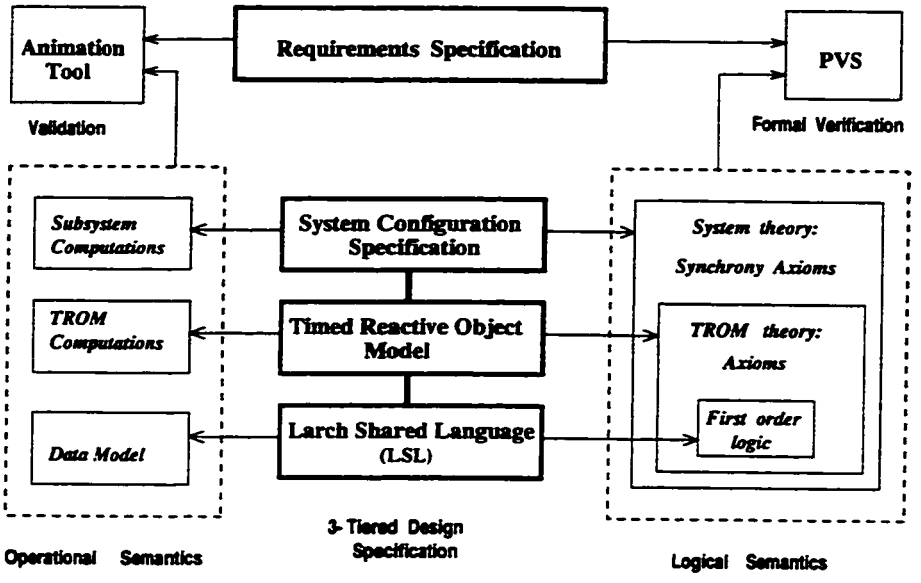


Figure 2: Overview of TROM methodology [Ach95].

specifies object collaborations, where each object is a TROM. The communication mechanism between TROM objects is based on *synchronous message passing* which is assumed to occur at a *port* associated with the TROM. Message passing involves an *event* and underscores an activity which takes an *atomic* interval of time. A *reactive* object modeled in TROM is assumed to have a single *thread of control.*

2.3.1 TROM Syntax

The structure and behavior of a TROM can be described either textually or visually. The TROM model incorporates the essential features for describing reactive entities. A TROM is a hierarchical finite state machine augmented with attributes, logical assertions on the attributes, and time constraints. The TROM object has a single thread of control, and communicates with its environment through ports, by synchronous message passing. The ports represent access points for bidirectional communication between the objects. The port-type of a port determines the messages that are allowed at the port. A TROM can have several port-types associated with it, and several ports of the same port-type. An event represents an activity taking an atomic interval of time, while an action represents an activity taking a non-atomic time interval of finite duration. At any instant, a TROM exhibits a signal representing either a message, an internal activity, or idleness. A signal describes the occurrence of an event at a specific time instant, at a specific port [Ach95].

Informally, an object defined in TROM consists of the following elements:

- A set of **events** partitioned into three sets: input, output and internal events. The

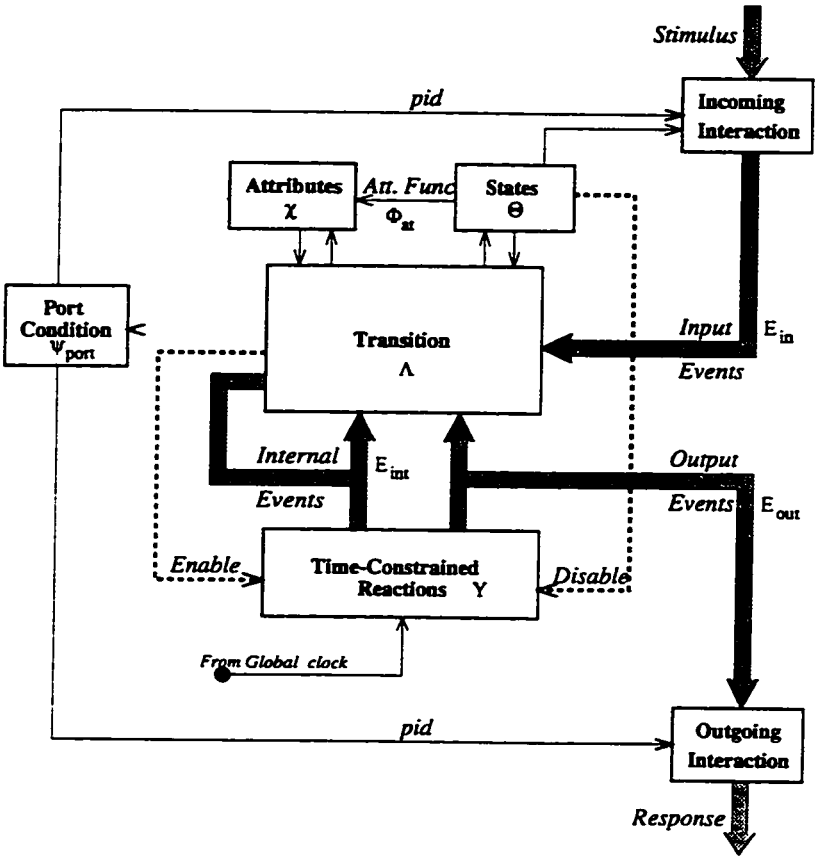


Figure 3: Components of a TROM [Ach95].

input and output events represent message passing and are suffixed by the symbols ? and !, respectively.

- A set of **states**: A state can have *substates*. An initial state is marked by the symbol *.
- A set of typed **attributes**: The attributes can be of one of the following two types: (i) an abstract data type signifying a *data model*; (ii) a port reference type.
- An **attribute-function** defining the association of attributes to states. For a computation associated with a transition entering a state, only the attributes associated with the state are modifiable and all other attributes will be *read-only*.
- A set of **transition specifications**: Each specification describes the computational step associated with the occurrence of an event. A transition specification has three assertions; a *pre-* and a *post-condition* as in Hoare logic, and a *port-condition* specifying the port at which the event can occur. The assertions may involve the attributes, and the keyword **pid** (port-identifier).

- A set of **time-constraints:** Each time constraint specifies the *reaction* associated with a transition. A reaction is the firing of an output or an internal event within a defined time period. Associated with a reaction is a set of *disabling states.* An *enabled* reaction is disabled when the object enters any of the disabling states of the reaction.

Figure 3 [Ach95] illustrates the components of a TROM. An external stimulus instigating an incoming interaction results in an input event at a port constrained by the port-condition for the transition. Every event gives rise to a computational step, updating the state of the object and its attributes. Only active attributes, as determined by the attribute functions, can be updated by the computation. A computational step may result in the enabling of a time-constrained reaction, the disabling of an outstanding reaction, and the firing of an outstanding reaction in the form of a transition. The firing of a reaction may lead to the generation of an output event at a port specified by the port-condition.

Several forms of non-determinism are supported by the TROM model. *Control non-determinism* is supported by allowing valid choices in the transition to be fired in a given state. A port-condition can allow a choice in the port at which an event associated with the transition to be fired can occur. An object can thus select the entity with which it wants to interact from its environment, supporting *interaction non-determinism.* A time constraint specifies minimum and maximum time delays between the enabling of a reaction and its firing, allowing a range of time instants at which the reaction can occur; this supports *timing non-determinism.* The computation associated with a transition, as specified by the post-condition, can be involve non-deterministic constructs, supporting *computation non-determinism.*

The patterns of interactions between entities in a system can be specified by the port-conditions. Several real-time features are supported by the model, including minimal and maximal delays, exact occurrences, and periodicity of event occurrences. Timing constraints are encapsulated, and an input event cannot be constrained.

2.3.2 TROM Operational Semantics

The status of a TROM captures the state in which the TROM is at that instant, the value of the attributes at that instant as reflected in the *assignment vector*, and the timing behavior of the TROM as specified in the *reaction vector.* The reaction vector associates a set of *reaction windows* with each time constraint, where a reaction window represents an outstanding timing requirement to be satisfied by the output event or the internal event associated with the time constraint. When the reaction vector is null, the TROM is in a *stable status.*

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

A *computational step* [Ach95] of a TROM is an atomic step which takes the TROM from one status to its succeeding status as defined by the transition specifications. Every computational step of a TROM is associated with a transition in the TROM; and every transition is associated with either an interaction signal, or an internal signal, or a *silent* signal. A computational step occurs when the TROM receives a ***signal*** and there exists a transition specification such that the following conditions are satisfied: the triggering event for the transition is the event causing the signal; the TROM is in the source state or a substate of the source state of the transition specification; the port-condition is satisfied if the signal is an interaction; and the enabling condition is satisfied by the assignment vector. The effects of the computational step are: the TROM enters the destination state or the entry state of the destination state of the transition specification; the assignment vector is modified to satisfy the post-condition; and the reaction vector is modified to reflect the firing, disabling, and enabling of reactions. The status of a subsystem is the set of statuses of the TROM objects in the subsystem. The computation of a TROM is a sequence of computational steps.

Each computational step is associated with a transition in the state machine of the TROM. Any transition leaving the current state of the TROM object can define the computational step, provided the assignment vector satisfies the enabling condition. Thus, the source state of the transition can be the current state of the TROM, or a superstate of the current state. After the transition is taken, the current state will be the destination state of the transition, or its entry state if it is a complex state. The port at which an interaction occurs must satisfy the port-condition associated with the transition, thereby constraining the objects with which the TROM can interact at that instant.

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

semantics ensures that time cannot advance past a reaction window without either firing or disabling the associated outstanding reaction.

The behavior of a TROM is described by the infinite sequences of computational steps it can undergo. The computation of a TROM is a sequence of alternating statuses and signals, where the transition between each pair of successive statuses is described by a computational step. If the sequence of steps is finite, then the terminating status is a stable status. The time progresses as the computation proceeds.

The factors determining whether a TROM is well-formed are:

- There is at least one transition leaving every state, thus barring the TROM from having a final terminating state.
- If there is more than one transition leaving a state, then the enabling conditions of the transitions should be mutually exclusive.
- Before a TROM starts executing, the values of only the active attributes in the initial state are specified; the values of the dormant attributes are undefined. An attribute will acquire a value only when it reaches the first state in which it is active. It is therefore necessary to ensure that an attribute which is dormant in the initial state becomes active in some state before the attribute is used.
- Every computational step in a TROM results in some computation of the TROM.

2.3.3 TROM Logical Semantics

The logical semantics is expressed using a set of axioms that can be used to verify the requirements properties of a system. The axioms for a TROM object can be obtained by substituting its status information into the arguments of the axioms. The resulting set of axioms and the synchronization axioms are subsequently used in the verification process. We first state the axioms informally; next, we give formal descriptions for some of them, and then derive these axioms relevant to the *Train* object.

1. *Atomic-event axiom:*
   There can be at most one event occurring in a TROM at any time instant. Also, an event can occur only at one port at any time instant.
2. *Silent-event axiom:*
   The occurrence of the silent event *tick* at any instant precludes the occurrence of any other event in the TROM at that instant.

3. *State-hierarchy axiom:*

   When a TROM is in a substate of a state θ, the TROM is also in the state θ. Similarly, when a TROM is in a complex state θ, the TROM is also in at least one of the substates of θ.

4. *State-uniqueness axiom:*

   A TROM cannot be in more than one state at any instant, unless the states have a hierarchical relationship. Thus, a TROM can be in two states only if one state is a substate of the other.

5. *Initial-state axiom:*

   A TROM has a unique initial state which is atomic. At the initial instant, a TROM is in the initial atomic state.

6. *Initial-attribute axiom:*

   At the initial instant, the attributes of a TROM satisfy an initial formula. This initial assertion is the maximal property satisfied by the attributes at that instant, such that any other assertion satisfied by the attributes is implied by the initial assertion.

7. *Dormant-attribute axiom:*

   An attribute is dormant in a certain state if the attribute is in the complement of the set of attributes obtained from the attribute function for that state. The value of a dormant attribute in a certain state cannot be modified as long as the TROM is in that state.

8. *Occurrence axiom:*

   For an event e to occur at a port p_i of a TROM, the TROM must be in the source state of a transition labeled by the event e, such that the port-condition of the transition is satisfied by the port p_i, and the enabling condition is satisfied, at the time of occurrence of that event.

9. *Transition axiom:*

   The occurrence of an event results in a state transition to the destination state or the atomic entry state of the destination state, and the post-condition of the transition specification is satisfied in the destination state.

10. *Persistence axiom:*

    For each state of a TROM, when no event causing a transition to leave that state occurs, the TROM does not change state and the value of attributes active in that state does not change.

11. *Time constraint axioms:*

The reactive behavior of a TROM is defined by the following set of axioms.

(a) *Activation axiom:*

A reaction is activated when a transition triggering the reaction occurs.

(b) *Constrained-event axiom:*

A trigger event is necessary for the occurrence of a constrained event.

(c) *Enabling axiom:*

The necessary conditions for a reaction enabled at time t to remain enabled in the succeeding time t' are: (1)the constrained event should not occur at time t, since the occurrence of the constrained event will fire the already enabled reaction; (2) the reaction is not disabled at time t'.

(d) *Disabling axiom:*

An enabled reaction will no longer remain enabled if the constrained event of the reaction is disabled due to the TROM entering into some disabling state.

(e) *Firing axiom:*

An enabled reaction is fired by the occurrence of the constrained event. Since the firing of the reaction satisfies an enabled reaction, the reaction will no longer remain enabled.

(f) *Prohibition axiom:*

If a reaction is enabled then the constrained event should not occur during the minimum delay period from the time of activation.

(g) *Obligation axiom:*

If an enabled reaction is not disabled within the maximum time bound after the time of activation, then the constrained event should be fired at some time within the maximum time bound from the activation instant.

(h) *Validity axiom:*

A reaction involving a constrained event e can be enabled at time t only if the triggering event f has occurred at time t_a such that t is within the maximum time bound from the activation instant t_a.

2.3.4 Case Study: Railroad Crossing Controller

We illustrate TROMLAB capabilities through a generalized version of the *railroad crossing* problem [HL94]. In this section we illustrate the TROM formalism for this problem. In the

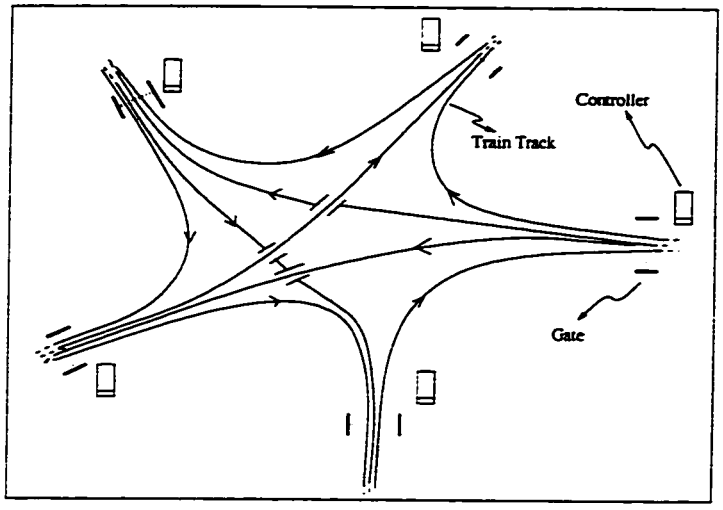


Figure 4: A complex railroad crossing system.

TROM *Train* [@*C*]
  Events: *Near*!*C*, *Exit*!*C*, *In*, *Out*
  State: ∗*S*1, *S*2, *S*3, *S*4
  Attributes: *cr*:@*C*
  Attribute-function:
    $S1, S3, S4 \mapsto \{\}$; $S2 \mapsto cr$;
  Transition Spec:
    $R_1 : \langle S1, S2\rangle$; *Near*!(*true*); *true* $\Longrightarrow$ $cr' = \text{pid}$;
    $R_2 : \langle S2, S3\rangle$; *In*; *true* $\Longrightarrow$ *true*;
    $R_3 : \langle S3, S4\rangle$; *Out*; *true* $\Longrightarrow$ *true*;
    $R_4 : \langle S4, S1\rangle$; *Exit*!(pid = *cr*); *true* $\Longrightarrow$ *true*;
  Time-constraints:
    $(R_1, In, [2, 4], \{\})$
    $(R_1, Exit, [0, 6], \{\})$
end

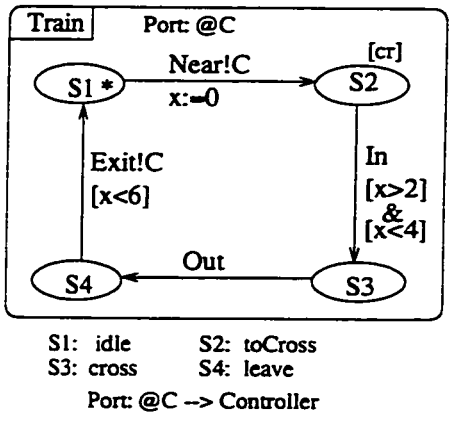


Figure 5: Class specifications for Train.

generalized version considered here, more than one train can cross a gate simultaneously, probably through multiple parallel tracks, as shown in Figure 4; a train can independently choose the gate it will cross, probably based on its destination. The entities interacting in the system are trains, controllers, and gates. The specifications of TROM classes and their state diagrams are shown in Figures 5, 6, and 7. The LSL trait describing the data model *Set* is available in [GH93].

A train sends the messages **Near** (**Exit**) to a controller indicating that it is approaching (exiting) the gate. The train triggers the internal event **In** within a window of 2 to 4 time units after sending the **Near** message, and sends the **Exit** message within 6 time units from sending the **Near** message. A controller sends the messages **Lower** (**Raise**) to the gate it is

TROM *Controller* [@P,@G]
  Events: $Near?P, Exit?P, Lower!G, Raise!G$
  State: $*C1, C2, C3, C4$
  Attributes: $inSet : PSet$;
  Traits: $Set[@P, PSet]$ /* Link to LSL tier */
  Attribute-function:
    $C1 \mapsto \{\}$; $C2, C3, C4 \mapsto \{inSet\}$;
  Transition Spec:
    $R_1 : \langle C1, C2\rangle$; $Near?(true)$;
      $true \Longrightarrow inSet' = insert(\text{pid}, inSet)$;
    $R_2 : \langle C2, C2\rangle, \langle C3, C3\rangle$; $Near?(\neg(\text{pid} \in inSet)$;
      $true \Longrightarrow inSet' = insert(\text{pid}, inSet)$;
    $R_3 : \langle C2, C3\rangle$; $Lower!(true)$; $true \Longrightarrow true$;
    $R_4 : \langle C3, C3\rangle$; $Exit?(\text{pid} \in inSet)$;
      $(size(inSet) > 1) \Longrightarrow inSet' = delete(\text{pid}, inSet)$;
    $R_5 : \langle C3, C4\rangle$: $Exit?(\text{pid} \in inset)$;
      $(size(inSet) = 1) \Longrightarrow inSet' = delete(\text{pid}, inSet)$;
    $R_6 : \langle C4, C1\rangle$; $Raise!(true)$; $true \Longrightarrow true$;
  Time-constraints:
    $(R_1, Lower, [0, 1], \{\})$
    $(R_5, Raise, [0, 1], \{\})$
end

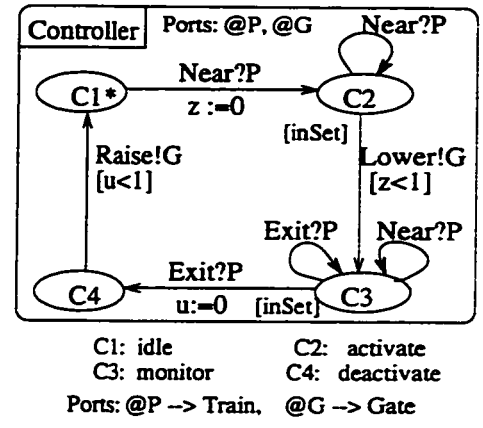


Figure 6: Class specifications for Controller.

controlling, indicating that the gate has to be lowered (raised). The Lower message is sent by the controller within 1 time unit from receiving the Near message while in the idle state: the Raise message is sent within 1 time unit from receiving the Exit message from the last train leaving the crossing. The gate triggers the internal event Down within 1 time unit from receiving the Lower message from the controller, and triggers the internal event Up within a window of 1 to 2 time units after receiving the Raise message. The safety requirement is that whenever a train is crossing a gate, the gate must be closed. In addition, the controller must be monitoring the gate at that point. For the system to operate properly, the gate must eventually be raised, and must remain so for a certain period of time before it is lowered again.

The *system configuration specification* defining the Train-Gate-Controller system is obtained by composing instances of the TROM classes, as shown in Figure 8. The Include section is optional, and is used to import other subsystems. A TROM object is defined in the Instantiate section; the cardinality of ports and the value of any active attribute in the initial state of the TROM is specified in the definition. The Configure section defines links between the different ports of the TROM objects in the subsystem and in the included subsystems, allowing interactions between the objects. Only *compatible* ports can be linked, such that an event sent at one port is acceptable as an input event at the other port at

TROM *Gate* [@*S*]
Events: *Lower?S*, *Raise?S*, *Down*, *Up*
State: $*G1, G2, G3, G4$
Transition Spec:
$R_1 : (G1, G2)$; *Lower?(true)*; *true* $\Longrightarrow$ *true*;
$R_2 : (G2, G3)$; *Down*; *true* $\Longrightarrow$ *true*;
$R_3 : (G3, G4)$; *Raise?(true)*; *true* $\Longrightarrow$ *true*;
$R_4 : (G4, G1)$; *Up*; *true* $\Longrightarrow$ *true*;
Time-constraints:
$(R_1, Down, [0.1], \{\})$
$(R_3, Up, [1, 2], \{\})$
end

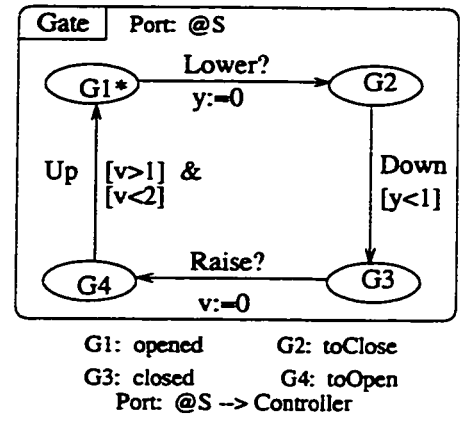


Figure 7: Class specifications for Gate.

SCS *TrainGateSystem*
Include:
Instantiate:
$A_1, \ldots, A_m$ ::*Train*[@$C : n$].*Create*()
$B_1, \ldots, B_n$ ::*Controller*[@$P : m$,@$G : 1$].*Create*()
$C_1, \ldots, C_n$ ::*Gate*[@$S : 1$].*Create*()
Configure:
$\forall i \in 1 \ldots m,\ j \in 1 \ldots n$
$A_i.@c_j$ — $B_j.@p_i$
$B_j.@g_1$ — $C_j.@s_1$
end

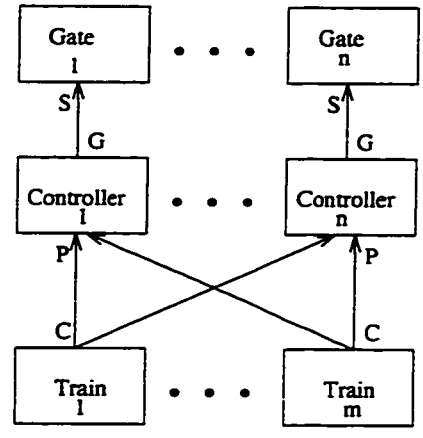


Figure 8: System configuration specification for Train-Gate-Controller system.

the same time. *One-one*, *one-many*, and *many-many* relationships between objects of the TROM classes can be specified. For instance, there is a many-many relationship between trains and controllers, and a one-one relationship between controllers and gates.

A system configured with two trains, two controllers and two gates will have the following ports and links: each train will have two ports of the same type referring to controller; each controller will have two ports of the same type referring to train and one port referring to gate; each gate will have one port referring to controller; one gate is linked to only one controller and this is achieved by linking the unique port of the gate to that port of controller referring to gate; each train is linked to both controllers and this is achieved by linking the two ports of that train to one port referring to train in each controller. Thus, the system configuration description is succinct and expressive.

Let us consider one step of computation of the subsystem with two trains, one controller and one gate. When there are no events, no train is crossing a gate and hence the controller

is idle and the gate is open. From the model shown in Figures 5, 6, and 7, notice that a gate is open until it receives the message **Lower** from the controller. The message **Lower** is output by the controller only after the message **Near** is received from a train. That is, the whole system remains **idle** until a train approaches the gate causing the output event **Near** to occur in the train. Now, the status of the system changes: the train is in state S2 (**to cross**), the controller is in state C2 (**activate**) and the gate is in state G1 (**Opened**). Suppose that within the next 2 time units, the other train outputs the message **Near**. In this case, during the next computational step, the value of the attribute **inSet** in state C2 (**activate**) or C3 (**monitor**) of the controller will reflect the fact that there is one more train to cross. The state of the controller will then be either C2 or C3 (depending on whether the message **Lower** has been sent to the gate), and the state of the gate will be either G1 (**opened**) (if the message **Lower** has not yet been received from the controller), or G2 (**toClose**) or G3 (**closed**) (if the message **Lower** has already been received). Notice that in this case the controller will be in the state C4 (**deactivate**) only when both trains will have sent the message **Exit** to the controller. The value of the attribute **inSet** is adjusted each time the controller receives a message from a train. The gate will receive the message **Raise** only when no train is in the crossing. The other scenarios and successive computational steps can be simulated following the operational semantics [Ach95].

2.4 Animation Tool

The specification environment includes a grammar [Tao96] supporting the formal description of TROM classes and system configurations. The editor facilitates the specification of a reactive system in the TROM formalism. The TROM specification of a reactive object is presented as a *class* definition. A class definition follows strictly the formal definition of TROM [Ach95]. Type-checking facilities are provided by a static analyzer [Tao96] incorporating an interpreter. The interpreter performs lexical and semantic analysis on the class definitions, and on the specification of system configurations. While parsing the input specifications, the interpreter constructs an internal representation of the data. This abstract syntax tree is subsequently used by the interpreter for semantic analysis, by the axiom generator to instantiate the axioms for a TROM object, and by the simulator to access the static components of a TROM object. The semantic analysis of the specifications allows a rigorous static inspection of the data types involved in the TROM classes and subsystems.

The development environment is supported by a simulator, an axiom generator, a verification manager and a graphical user interface. Animation of the simulation process is done

through the graphical user interface, allowing visualization of the state transitions. Debugging facilities include freezing the simulation and activating the validation tool. When simulation is frozen, the user can interact with the process to inject input events, and query the behavior of the system being simulated. The user can walk through the event trace and examine the history of the simulated scenario, roll back to an earlier instant in time and restart from that point. Due to environmental changes, requirements of a real-time reactive system may evolve throughout the life of a system. Registering requirements, relating requirements to objects that are affected by it, and knowing the relationship among the requirements are important to the development process. The user interface, which includes a browser, controls the visualization process for both the static and dynamic aspects of requirements, specifications, and objects. Besides providing the standard set of user interface functionalities, it will expose the visual images of TROM objects and their interactions during the simulation process. This part of the tool is under development, and hence we do not discuss it further.

The reactive formalism is such that incremental compilation of classes, although possible, becomes laborious. As such, after error correction or making changes to existing specifications, the specifications must be recompiled. Specifications for the data structures in the lowest tier, that is, LSL [GH93] traits, are compiled separately and checked for consistency and adequacy using Larch syntax analyzer. The traits are stored in a library and can be imported by TROM classes. TROM classes, forming the middle tier, are compiled individually, and independently of the other tiers. Adding a new state, event, attribute, transition specification, or time constraint can be done incrementally, involving static and semantic analysis of the relevant class only. Redefining the system configuration from the upper-most tier involves a recompilation of the *System Configuration Specification* (SCS) only; the specifications from the other tiers need not be recompiled in such cases.

2.5 Formal Verification Methodology

The axiom generator uses the logical semantics of the TROM objects and the system configuration to generate a set of axioms for the specific problem being simulated. This set of axioms may have to be supplemented by additional axioms on the temporal properties of time intervals. At any instant during simulation, the set of axioms can be instantiated by copying the status of the system into the arguments of the axioms. This will reduce the axioms to a set of propositions, which can then be reduced in a decidable way to produce an answer to a query. These axioms can thus be used by the simulator for on-line analysis of system behavior. TROMLAB environment supports formal verification to be done

interactively. When this phase is completed, the axioms generated by the axiom generator can be reduced to propositions and used to deduce whether or not a property stated as a proposition at an instant is a consequence of the history of system status. The user must build a set of sufficient axioms, and follow the strategy explained in Chapter 5.

The verification methodology allows proving that a TROM-based system satisfies a property. Axioms are derived from the class specifications to describe the transition specifications, and the time constraints. We include other axioms to specify synchronized messages from one TROM object to another. The property is expressed as an invariance on a logical assertion about the state of the system. The axioms are translated into inequalities over absolute time variables. We include lemmas over the same variables to express the invariance assertions. We then prove the lemmas using the axioms expressed in absolute times.

2.6 Related Work

The building blocks for developing reactive systems in TROMLAB environment come from TROM formalism [Ach95]. Timed extensions of IO automaton such as Time Constrained Automata [TMM88] and TRA [Bes91] provided the basic inspiration for the work on TROM. However, there are some key differences: (1) TROM is grounded on very specific structural framework built on object-oriented (OO) paradigm; (2) TROM is not restricted to design systems that are only input-enabled; and (3) The notion of hierarchical states, associating attributes (and their abstract types) to abstract states through attribute function, and inheritance and subtyping are novel and new in TROM.

The other works related to state-based modeling of reactive systems are Objchart [GM93], Disco [JKSSS90], TRIO++ [MSP94], and ROOM [SGW94]. Among them, ROOM supports OO in a true sense and comes close to our modeling approach. Disco and Objchart do not support specification of real-time constraints. TRIO++ emphasizes expressing the requirements specification of real-time systems using OO principles. However, the language lacks important OO concepts such as subtyping relationships between classes, the notion of concurrency and message passing between objects, and above all lacks the facility to describe system models. In spite of the apparent closeness of our model to ROOM, which is based on *Actors* [Agh86], there are important differences: (1) ROOM allows the specification of only two types of timing constraints: *latency*, and *service* times. These constraints are more biased towards an implementation. Specification related timing constraints such as stimulus-response, response-response constraints cannot be specified. (2) A major drawback of ROOM is its restricted applicability to input-enabled systems [LT87]. (3) Operation descriptions in ROOM use concrete state variables. Consequently, assignments to them are

biased to implementation. (4) There is no data abstraction facility in ROOM and no formal semantics is given. Consequently, rigorous validation and verification methods do not exist for reactive systems implemented under ROOM methodology.

Recognizing that tools are essential to comprehend the behavior of complex systems, several tools have been proposed for the development of reactive systems. The two notable ones are STATEMATE [HLN+90] and SIP [FS93]. STATEMATE is based on the *statechart* [Har87] formalism. The formal semantics of statecharts allows execution of a system specified using a statechart. The tool uses graphic displays to show the transformations of the statechart during simulation; it also incorporates debugging functionalities. SIP simulates the behavior of reactive systems specified using statecharts; SIP uses the reasoning system FRAPPE to deduce answers to questions about the behavior of the system during simulation. The development tool in TROMLAB environment differs from the above two tools in several important ways:

- Our tool supports an object-oriented approach to reactive system development. None of the above systems reap the full benefits of the object-oriented and iterative approaches to software development.
- The TROM object-oriented methodology has a formal operational and logical semantics, thus promoting validation and verification.
- The design specification supported by the tool is three-tiered: the top most tier constitutes *System Configuration Specification (SCS)*, which describes object interactions; the middle tier gives the detailed specification of the objects extracted from the problem domain as described in the requirements; the lowest tier specifies the data abstractions used in the class definitions of the middle tier through *Larch Shared Language*, one of the languages of Larch [GH93].

The significant differences among TROMLAB, ObjecTime, STATEMATE, and SIP are shown in Table 1.

	Object-orientation	Data abstraction	Simulates execution of
ObjecTime	√	–	implementation
Statemate	–	–	formal specification
SIP	–	–	formal specification
TROMLAB	√	√	formal specification

Table 1: Tools supporting development of reactive systems.

Chapter 3

Animation Tool

A major issue in designing reactive systems is understanding the behavior of the system under development. The complexity of reactive systems makes an enumeration of all possible scenarios in the behavior of such a system impractical. Simulation allows observing the behavior of a system through trace analysis of the simulated scenarios. Animation provides a graphic visualization of interactions among the entities. An important goal of animating the simulation process is to facilitate design-time debugging, and validation of system design against requirements.

3.1 Introduction

This chapter describes the functionalities and design of the animation tool. Section 3.2 outlines the simulation process, and the experimentation, debugging, and validation facilities for TROM-based systems. It includes outlines of the analysis process for simulated scenarios, the analysis of static components, and feedback on object behavior. Section 3.3 sketches the design of the tool, including its architecture and components. Section 3.4 describes the modules designed to support each of the three tiers in the TROM methodology, and its operational semantics.

3.2 Functionalities

The basic operation is the simulation of a computational step of the system built on TROMs. The necessary condition is that the events appear in non-decreasing order of their activation times in the *simulation event list.* The event list is the backbone of the simulator. During the simulation process, the clock can be frozen before handling the next event from the event list. The state of the system at this point and the histories accumulated during the simulation of computational steps up to this point can be inspected, and analyzed for the causes of the current system status. The user may use this information to correct errors in the design of the TROMs and the subsystem. The user can make changes to the *not yet simulated* aspects of the system, such as modifying transition specifications or timing constraints. Under certain conditions, the simulation can be restarted from the latest frozen clock instant without recompilation of the TROM specifications.

To contain complexity, the user may interactively conduct design validation by simulating a single computational step and analyzing the causes of unexpected results. By going backwards through the history maintained by the simulator, the user can debug the design making use of the requirements and the knowledge about the expected behavior. This is in contrast to using an automated reasoning system as in SIP [FS93]. To support design evolution, events can be injected interactively during simulation; logical assertions on the transition specifications can be specified to simulate a scenario. At any time, the user can roll back to analyze the evolution of system status and simulate corrections in the design.

3.2.1 Simulation

The simulator can be activated only if the specifications have been successfully type-checked. The *pace* of the simulation process can be set to a slow mode so that systems where events occur within fractions of a second can be simulated. Similarly, the pace can be increased

so that systems where events occur at a slow pace can be simulated. Thus, the user can experiment with the system under development to ensure proper and timely scheduling of events. The facilities also allow the user to ensure that deadlines are met and race conditions avoided by the system. Thus, a system can be validated to satisfy stringent timing requirements.

While simulating a system, the user can interact with the toolset to inject a simulation event, and analyze its consequences. Similarly, the histories of the simulation events store sufficient information to allow the user to roll-back through the simulation process, and examine the trace of the simulated scenario, or to continue the simulation from a certain point in time. Facilities are included to allow the user to analyze the trace of the scenario. The user can examine the status of the system or of TROM objects in the system, at any point in the trace of the scenario. At any point in time, the user can freeze the clock and activate the validation tool to inspect the status of the system and the events in the simulation event list.

3.2.2 Design Debugging and System Validation

Timing constraints stated in the requirements are mapped to design without knowledge about their consistency. Inconsistent timing constraints can lead to deadlock configurations and violation of safety and liveness properties. The simulation process does not algorithmically detect a deadlock. However, if no input event has been scheduled for a pre-determined period of time, we can draw exactly one of the following conclusions by examining the status of the TROMs at this point:

- the reaction vector is a null vector, and all the events in the event-list have been handled; the system has entered a *stable state*, and all timing constraints have been satisfied;

- the system cannot progress, since two synchronized TROMs are mutually waiting for each other to enable an event causing the next transition to occur; the state of the event-list and the state of the reaction vector for the TROMs in the system are irrelevant in this case; the configuration of the system introduces deadlock.

When a deadlock configuration is identified, the user iterates through the validation process to redesign the formal model of the software unit, until the behavior of the system conforms to the requirements.

At each step in the simulation, the satisfaction of safety properties at that instant can be checked. If the safety properties continue to hold at successive simulation steps until

the system reaches its stable state, the user can infer with some confidence that the system is safe over the histories simulated so far. However, the system cannot be declared to be absolutely safe, since safety properties have not been verified for all possible scenarios. To demonstrate that a safety property holds for all scenarios, we need the help of a theorem prover. When there is violation of a safety property at a simulation step, the history of the computational steps can be analyzed to determine the causes of error. The design must then be corrected before restarting the simulation. Incremental changes to timing constraints that deviate from their original specifications but retain safety property may also be attempted as part of the simulation process.

Debugging Facilities

Debugging facilities provided include the following operations:

- continue the simulation process,
- display the current simulation time,
- display the status of the system,
- display the status of a subsystem,
- display the status of a TROM,
- display the simulation event list,
- inject a simulation event,
- roll-back to a given point in time,
- activate the *query handler*,
- activate the *trace analyzer*,
- terminate the simulation process.

Trace Analysis of Simulation Scenario

The facilities provided for trace analysis of simulation scenarios include displaying the following dynamic information:

- simulation events which have triggered a transition,
- simulation events which have not triggered a transition,

- simulation events which have not yet been handled,
- simulation events which have occurred during a certain period,
- simulation events which have triggered a transition for a given TROM.
- simulation events which have triggered a transition for a given TROM during a certain period,
- status of the system simulated at a given point in time,
- status of a subsystem at a given point in time,
- status of a TROM in the system at a given point in time.

Analysis of Static Components during Simulation

Query handling facilities provided include functions to display the following *static* information about a specified TROM:

- Abstract Syntax Tree,
- transition specifications,
- transition specifications with its current state as source,
- transition specifications with a given state as source,
- transition specifications with a given state as destination,
- transition specifications triggered by a given event,
- time constraints,
- time constraints for a given triggering event,
- time constraints for a given constrained event,

In addition, the user is allowed to terminate the simulation process at any point in time, and also to switch between the simulation and debugging processes. The specifications for the Train-Gate-Controller system, and a sample run of the simulation process for the system, are included in Appendix C.

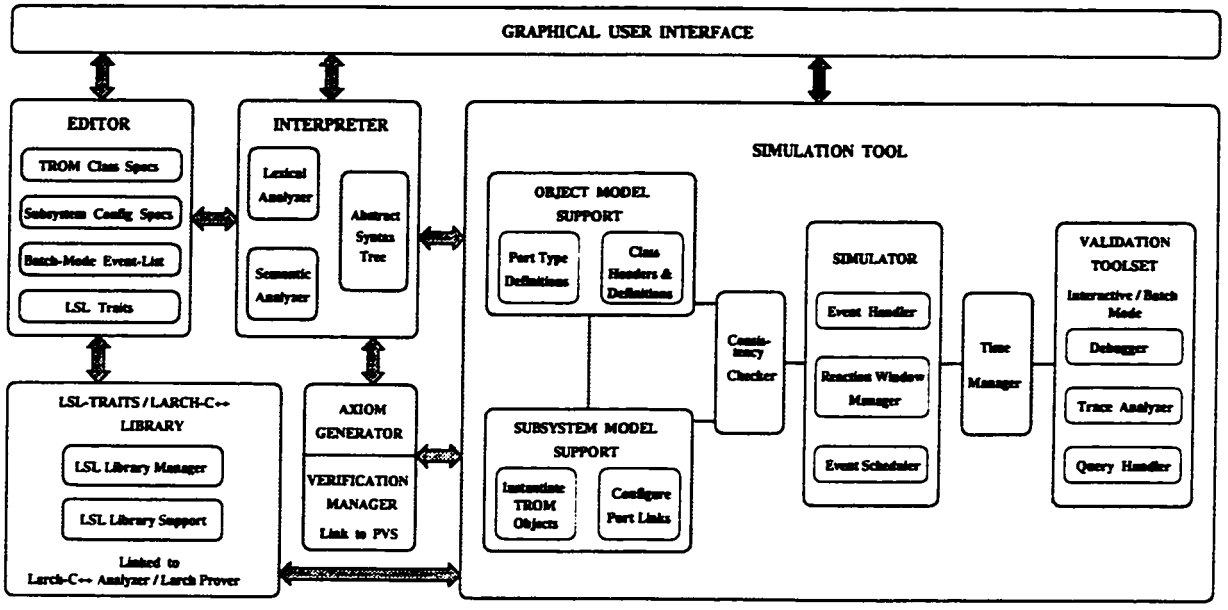


Figure 9: Structure of animation tool.

3.3 Design

The design of the animation tool is object-oriented; several objects interact with each other to simulate TROM-based systems. The main components include an editor, an interpreter, a graphical user interface, an axiom generator, and a simulator incorporating a validation tool. Objects in the simulator are designed to support the three-tiers of the TROM formalism, namely LSL traits, object models, and system configurations, as well as the operational semantics on which the simulator is built. Objects composing the validation tool support trace analysis of simulation scenarios, query handling, and debugging. Other objects are used for time management and consistency checking.

3.3.1 Object-Oriented Architecture

The design components of the tool are shown in Figure 9. To conform faithfully to the TROM formalism, the design of the tool is based on object-oriented principles. The interpreter, with the use of the editor, permits static analysis of TROM classes that are input to the simulator. The interpreter also constructs an internal representation of each TROM class and the subsystem configuration. It is the user's responsibility to prepare the specifications of TROM classes and subsystems, and their refinements according to the grammar on which the syntax analyzer is built. The user interface provides the necessary help to link the textual descriptions to their *visual* equivalents.

The simulation toolset consists of a *simulator*, a *time manager*, a *consistency checker*, a *validation tool*, and other objects to support the TROM formalism. During the simulation process, the simulator interacts with the *LSLLibraryManager* and the *LSLLibrarySupport* to create instances of traits defined in the LSL library, and to evaluate functions introduced in the traits. The simulator also interacts with the *axiom generator* to instantiate the axioms for a TROM object according to the logical semantics. The simulator "decorates" the Abstract Syntax Tree before starting simulation so that data connected to an aspect of the tree can easily be accessed through pointers during the process. It uses the data stored in the Abstract Syntax Tree for the class specifications and system configuration specifications to create and initialize the system to be simulated. The simulator also incorporates member functions providing access to the status of the system during simulation. These functions will be used by the graphical user interface to derive time charts portraying the behavior of the system in the simulated scenario.

3.3.2 Simulator Model

Objects in the *dynamic model* of the simulator include an *event handler*, a *reaction window manager*, an *event scheduler*, a *consistency checker*, and a *time manager*. The event handler takes an event which is due to occur and detects a transition which the event will trigger, taking into consideration the status of the TROM object receiving the event and the enabling and port conditions of the transition specifications. The event handler then updates the status of the relevant TROM object, and passes control to the reaction window manager. This module will actuate the computational step to handle the transition, thus causing reactions to be fired, disabled, and enabled. Whenever a reaction is enabled or disabled it sends a message to the event scheduler to schedule or unschedule a corresponding event. The event scheduler causes an enabled event to occur at a random time within the corresponding reaction window. It schedules output events through the *least recently used* port, using a *round-robin* algorithm. It also synchronizes input events with corresponding output messages directed towards TROM objects linked through ports.

The consistency checker ensures the continuous flow of interactions by detecting deadlock configurations. It uses a time-stamp on the occurrence of simulation events to identify a possible cycle in the sequence of transitions allowed by the specifications. The time manager maintains the simulation clock, updating it regularly. It allows setting the pace of the clock to suit the needs of analysis of simulation scenarios. It also allows freezing the clock while analyzing the consequences of a computation.

3.3.3 Validation Tool

The validation tool consists of a *debugger*, a *trace analyzer*, and a *query handler*. The debugger supports system experimentation by allowing the user to examine the evolution of the status of the system throughout the simulation process. It also supports interactive injection of a simulation event at any point in the simulation process, and simulation roll-back to a specified point in time. The trace analyzer includes facilities for a thorough analysis of the simulation scenario. It gives feedback on the evolution of the status of the objects in the system, and on the outcome of the simulation events. Thus, the behavior of the system being simulated can be analyzed periodically. The query handler allows examining the data in the Abstract Syntax Tree for the TROM class to which the object belongs, supporting analysis of the static components during simulation.

The simulation can be run in debugging mode, in which case the process is frozen when all the events occurring at a point in time have been handled. At this point the user is allowed to activate the debugger, to continue the simulation, or to terminate the process. When the debugger is activated, the user can examine the status of the system, or the individual statuses of the TROM objects, inspect the data stored in the Abstract Syntax Tree, and examine the trace of the simulated scenario. Functionalities include inspecting the status of the system at specific points in time, and the status of the simulation event list. The user can inject an event to observe the behavior of the system during simulation. or roll-back to a point in time, and continue the process from that point.

3.4 Supporting TROM

The design of the simulator supports the features of the TROM formalism, and faithfully conforms to the operational semantics. The *ObjectModelSupport* module supports the specification of TROM classes and the evaluation of logical assertions included in the transition specifications. The *SubsystemModelSupport* module creates subsystems by instantiating included subsystems, TROM objects, and port links. It also initializes the statuses of the TROM objects and defines the lists of ports for each object. The *LSLLibraryManager* supports the creation of instances of data structures defined as traits in the LSL library. The *LSLLibrarySupport* provides facilities for calling functions defined in the traits, and for their execution. This forms the operational implementation of LSL traits.

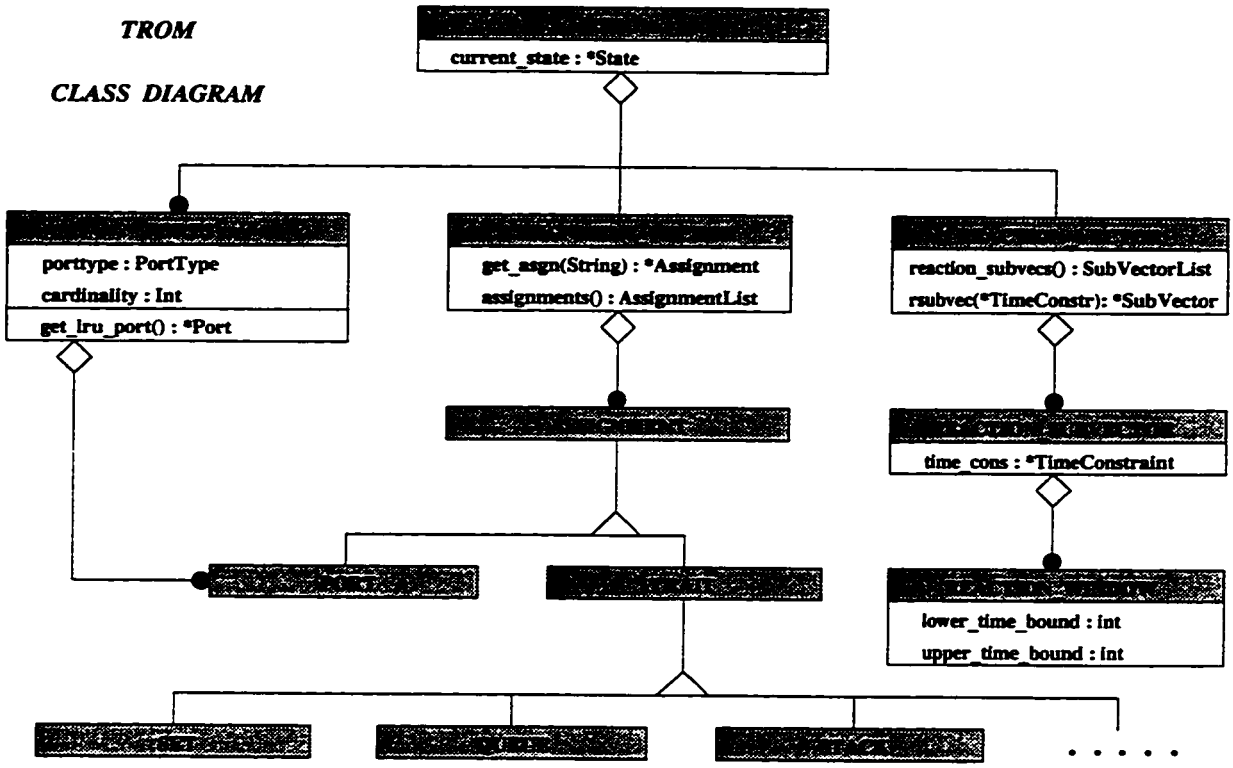


Figure 10: Class diagram describing TROM object model.

3.4.1 Object Model Support

To contain the complexity of the design of the simulator, *dynamic* data is separated from *static* data. The *status* of a TROM, comprising of its *current state*, *assignment vector* and *reaction vector*, constitutes dynamic information specific to the TROM object. These information, together with the port lists of the TROM object, are captured by the data members of a class from which all TROM objects are created. Static data representing information shared by all TROM objects of a particular class include sets of port types, events, states, attributes, transition specifications, time constraints, attribute functions, and LSL traits. These are encapsulated in a class definition; each TROM class is described by an instance of this class, representing the Abstract Syntax Tree for the TROM class. The tree is used to instantiate TROM objects, and to access the static data during simulation.

Every object of a TROM class has access to the corresponding instance of the Abstract Syntax Tree. Thus TROM objects of the same class share one copy of logical assertions on attributes and timing constraints on reactions. Encapsulating the characteristics of a TROM in a class supports inheritance, allowing a TROM class to inherit the characteristics of another TROM class. Figure 10 shows the class structure for the TROM, using OMT notation [RBP+91]. Figure 22 of Appendix A gives a detailed class diagram for TROM.

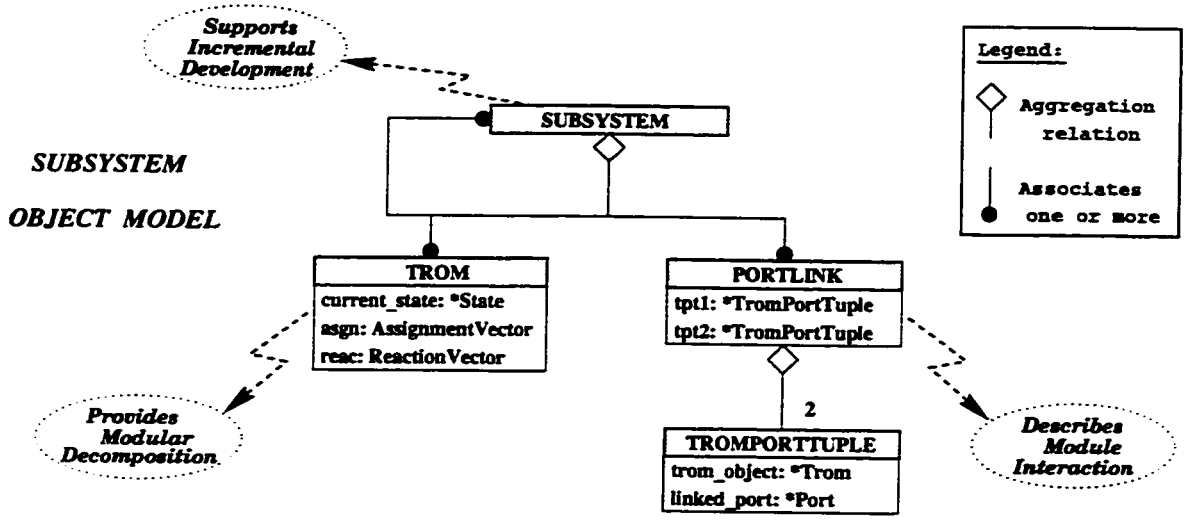


Figure 11: Class pattern for complex system development.

3.4.2 Subsystem Model Support

The *Subsystem* Model includes a set of TROM objects, a set of *included* subsystems, and a set of *port links* to configure interaction between the TROM objects. These data are encapsulated in the Subsystem class using *recursive aggregation* for the included subsystems. Port links are established between TROM objects in the subsystem and with TROM objects in included subsystems. This supports incremental development of larger subsystems using available components. Figure 11 shows the class structure for the Subsystem model, using OMT notation [RBP+91]. Figure 23 included in Appendix A gives a detailed class diagram for the Subsystem model.

We use recursive aggregation to support the inclusion of other subsystems in a subsystem, so that each included subsystem has its own sets of TROM objects, included subsystems, and port links. As supported by the formalism, TROM objects that are part of different subsystems in a system, can be linked through port links. A port link is defined as a pair of TROM-port tuples, where each tuple points to the TROM object, and the port through which it is linked. Recursive aggregation of subsystems supports the incremental development of the system, while the TROM objects provide modular decomposition, and the port links provide module interaction. Subsystems can be simulated independently of each other before being incorporated into a larger system.

3.4.3 LSL Trait Specification Support

The lowest-tier in the TROM methodology supports the inclusion of abstract data structures in the TROM class descriptions. This feature is supported in the simulation toolset by an

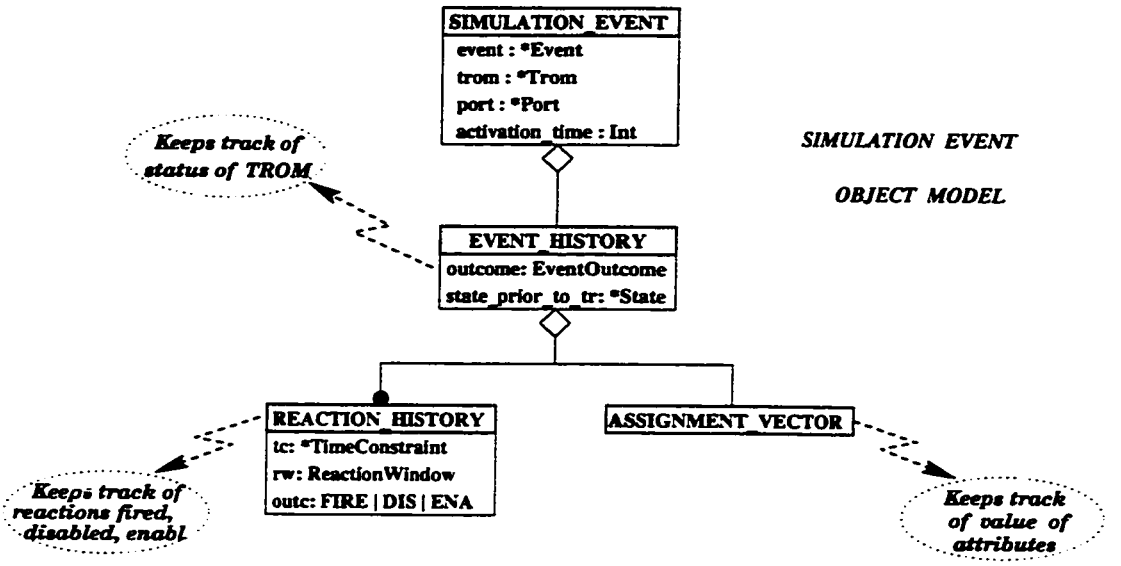


Figure 12: Class diagram for Simulation Event object model.

LSL library defining several traits representing data structures. The traits are initially defined in *Larch Shared Language*, and analyzed for consistency using the Larch syntax checker. The specifications can be translated into Larch/C++ class interface specifications before deriving an implementation. A TROM class can import a trait included in the library to specify an abstract data structure.

The LSL library can be augmented with new trait definitions, whenever new data representations are required by a system. The two objects, *LSLLibraryManager* and *LSLLibrarySupport* need to be maintained so that each new definition included in the library is properly represented. The LSLLibraryManager and LSLLibrarySupport allow the creation of instances of the traits, and the evaluation of functions introduced by the traits. This generic methodology supports the definition and execution of any abstract data structure which may be required in the system under development. The behavior of these data structures can thus be better understood.

3.4.4 Simulation Based on TROM Operational Semantics

While simulation goes forward in time, debugging requires that past actions and their effects be saved for analysis. Keeping this in mind, we have modeled the *Simulation Event* as shown in Figure 12, where each simulation event is associated with the computational step that it causes. This enables us to roll back to any previous state of the system, and examine the status of the TROM objects. Figure 24 included in Appendix A gives a detailed class diagram for the Subsystem model.

The occurrence of a simulation event causes a computational step in the computation of a TROM as defined by Achuthan [Ach95]. The simulator handles the event so that the status of the TROM object to which the event is directed, is modified in compliance with the computational step it is causing. Thus, the statuses of the objects in the system reflect the computation of the TROMs. In Chapter 4, we describe the simulation algorithm, and its conformance to the operational semantics of TROM.

Chapter 4

Formal Specification of Simulator

The VDM specification language includes a wide array of constructs in its grammar, supporting refinement of specifications. A formal specification of the functionalities of a software system such as the simulator provides insight into the intrinsic functions of the system. Errors in the design can be detected at an early stage, and an implementation of the specifications in a high-level language can be easily derived.

4.1 Introduction

We have specified the operations of the simulator in the VDM specification language [Gro95a]. In a first set of specifications, the operations are written as *implicit* operations. An expression and pre- and post-conditions express what is achieved by the operation. In a second version, most of the operations are rewritten as *explicit* operations, wherein a statement and pre- and post-conditions specify the behavior of the operation. The operations can be used to derive an implementation in a high-level language. Section 4.2 describes the algorithm used by the simulator. Section 4.3 outlines the steps followed in the formal specification exercise. Section 4.4 describes how this process has helped us to validate the functional model of the simulator, so that it faithfully conforms to the TROM operational semantics.

4.2 Simulation Algorithm

The algorithm given in Figure 13 represents the steps in the simulation of a system. The TROM specifications are first type-checked, and the classes pre-processed. If the specifications are syntactically and semantically correct, we create the system to be simulated. This involves instantiating TROM objects, subsystems, and port links. We then initialize the status of the system, giving initial values to the attributes of the TROM objects. The next step is to schedule internal events which originate from the initial state.

At this point, the simulator starts to handle the simulation events from the simulation event list. As long as there is no event whose occurrence time is equal to the value of the global clock, we increment the clock and do nothing. When a simulation event is due for handling, the simulator gets the corresponding transition specification, and handles the transition accordingly. It saves the current state of the TROM object and its assignment vector in the history of the simulation event. It then handles the transition according to the "handle-transition" operation given in Figure 16. The transition can cause reactions to be fired, disabled, or enabled, according to the semantics of the computational step.

```
begin /* simulation algorithm */
    type-check TROM class and subsystem specifications
    preprocess TROM classes to be used in simulation
    get label of Subsystem s to simulate
    instantiate Subsystem s
    instantiate Subsystems included in Subsystem s
    instantiate TROM objects for each Subsystem
    create PortList for each PortType for each TROM object according to port cardinality
    initialize current state and assignment vector of each TROM object
    configure PortLinks for each Subsystem
    initialize SimulationClock
    schedule unconstrained internal events from initial state for each TROM object
    for all SimulationEvent se in SimulationEventList sel
    begin /* at this stage SimulationClock can be frozen and debugger activated */
        while SimulationClock < occur time of SimulationEvent se
        begin
            increment SimulationClock /* using machine clock */
        end
        while exists SimulationEvent se and SimulationClock == occur time of se
        begin /* handle SimulationEvent se */
            get TROM object trom accepting SimulationEvent se from Subsystem s
            get TransitionSpec ts triggered by SimulationEvent se
                /* update history of SimulationEvent se */
            save current state of TROM object trom in EventHistory of se
            save assignment vector of TROM object trom in EventHistory of se
                /* update status of TROM object trom */
            change current state of TROM object trom to destination of TransitionSpec ts
                or to entry state of destination state of TransitionSpec ts if a complex state
            change assignment vector of TROM object trom according to postcondition of ts
                /* handle transition specified by TransitionSpec ts */
            for all TimeConstraint tc in list of TimeConstraints for TROM object trom
            begin
                if constrained event of TimeConstraint tc == label of SimulationEvent se
                begin
                    for each ReactionWindow rw in reaction subvector associated with tc
                    begin
                        if SimulationEvent se occurs within ReactionWindow rw
                        begin /* fire reaction according to TimeConstraint tc */
                            remove ReactionWindow rw from reaction subvector ass. with tc
                            insert ReactionHistory rh in EventHistory of se according to rw
```

```
                    end
                end
            end
            if current state of TROM object trom is in set of disabling states of tc
            begin /* disable reaction according to TimeConstraint tc */
                for all ReactionWindows rw in reaction subvector associated with tc
                begin
                    remove ReactionWindow rw from reaction subvector assoc. with tc
                    insert ReactionHistory rh in EventHistory of se according to rw
                    unschedule disabled SimulationEvent in SimulationEventList sel
                    if constrained event of TimeConstraint tc is an output event
                    begin
                        remove disabled SimulationEvent scheduled for synchronization
                    end
                end
            end
            if label of TransitionSpec ts == transition label of TimeConstraint tc
            begin /* enable reaction according to TimeConstraint tc */
                insert new ReactionWindow rw in reaction subvector associated with tc
                insert ReactionHistory rh in EventHistory of se according to rw
                    /* schedule new simulation event - internal/output event */
                insert new SimulationEvent se2 in SimulationEventList sel
                    using LRU Port of PortType of constrained event of tc
                    and random time within ReactionWindow rw
            end
        end
        schedule unconstrained internal event from current state for TROM Object trom
        if constrained event of TimeConstraint tc is an output event
        begin /* identify linked TROM object for synchronization */
            get PortLink pl from Subsystem s linking the two TROM objects
                /* schedule new simulation event - input event */
            insert new SimulationEvent se3 in SimulationEventList sel,
                using portlink pl, for synchronization
        end
        get next SimulationEvent se from SimulationEventList sel
    end
  end
end /* simulation algorithm */
```

Figure 13: Simulation algorithm implementing TROM operational semantics.

4.3 VDM Specifications

The formal specification process has helped us to clearly understand the functionalities of the simulator and derive programs based on the specifications. The data types in VDM correspond to the TROM components defined in Section 2.3. The semantics of the TROM model are captured by invariants on data types and on the *state* of the system. Based on the data types, the VDM state of the simulation model is defined to include the subsystem being simulated, a sequence of simulation events, a set of LSL trait definitions, and a global clock. The *functional model* of the simulator is specified by a number of VDM operations and functions. A full description of the VDM-SL specifications is included in Appendix B. The specifications are available from IFAD VDM examples repository [AM96b]. The formal specification of the simulator has provided insight into its functionalities, allowing us to eliminate errors in its design.

4.3.1 Data Types and TROM Representation

The data types described in the specification represent the components of the TROM model. They have been designed to conform to the model, so that an implementation can be derived directly from the specifications. Each component of a TROM is specified as a data type; similarly, each component of a system configuration specification is specified as a data type. Invariants are included to allow semantic analysis of the specifications. Figure 14 shows the data type for *Time Constraint*, and its invariant.

```
TimeConstraint::label : String
                transition : TransitionSpec
                constrainedevent : Event
                timebounds : ReactionWindow
                disablingstates : State-set
                reactionwindows : ReactionWindow-set

inv mk-TimeConstraint(label,transition,cevent,tbounds,dstates,rwindows) ≜
  (((cevent.type = INTERNAL) ∨ (cevent.type = OUTPUT)) ∧
   (∀ rw ∈ rwindows ·
       ((rw.uppertimebound - rw.lowertimebound) =
        (tbounds.uppertimebound - tbounds.lowertimebound))));
```

Figure 14: VDM data type specification for a time constraint.

```
state system of
    SUBSYSTEM : Subsystem
    SIMULATIONEVENTLIST : SimulationEvent*
    LSLLIBRARY : LSLTraitDefinition-set
    CLOCK : N
end
```

Figure 15: VDM state of the simulation model.

4.3.2 State of the Model and its Invariants

The state of the model include the system being simulated, a global clock, a sequence of simulation of events, and a set of LSL trait definitions. The invariants on the state of the system specify the semantics of the TROM formalism. Based on the data types, the VDM state of the simulation model is defined as shown in Figure 15.

The state of the model includes the system being simulated, a sequence of simulation events, a set of **LSL** trait definitions, and a global clock. The invariants on the state of the model assert the following conditions:

- there exists TROM objects in the subsystem or in the included subsystems,
- there exists port links, to allow communication between the TROM objects,
- the event-list is ordered according to the activation time of the events,
- there can be only one event accepted by a TROM at any time instant (single thread of control),
- any event in the list is accepted by a single TROM,
- each **LSL** trait used by a TROM is defined in the set of **LSL** trait definitions,
- the event-list is initially non-empty, and
- the global clock is initially 0.

4.3.3 Functions and Operations

The *functional model* of the simulation tool is specified by a number of VDM operations and functions. The *simulator* operation initializes the global clock, schedules internal events which are not time-constrained and which trigger transitions from the initial state, and handles all the events in the simulation event list. It freezes the clock to handle all the

```
handle-transition: Trom × SimulationEvent × TransitionSpec ⟶ ()
handle-transition(trom, se, ts) △
  (for all tc ∈ trom.timeconstraints
   do (if tc.constrainedevent.label = se.eventlabel
       then (for all rw ∈ tc.reactionwindows
             do (if se.occurtime ≥ rw.lowertimebound ∧
                    se.occurtime ≤ rw.uppertimebound
                 then (update-history-fire-reaction(trom, se, tc, rw);
                       fire-reaction(trom, se, tc, rw) )
                 else skip))
       else skip;
       if trom.currentstate ∈ tc.disablingstates
       then (for all rw ∈ tc.reactionwindows
             do (update-history-disable-reaction(trom, se, tc, rw);
                 disable-reaction(trom, se, tc, rw) ))
       else skip;
       if ts.label =tc.transition.label
       then (update-history-enable-reaction(trom, se. tc, ts);
             enable-reaction(trom. se, tc, ts) )
       else skip ))
  pre (se.occurtime = CLOCK)
  post (CLOCK = CLOCK);
```

Figure 16: VDM specifications for "handle-transition" operation.

events due at that time, and then *ticks* the clock until the next event needs to be activated. The *handle-transition* operation *fires* reactions which have been previously enabled, *disables* reactions associated with a time constraint when the TROM enters a disabling state, and *enables* reactions caused by the simulation event.

The functions include procedures which do not modify the state of the model, while the operations include procedures which modify the state of the model. The functional model of the simulator includes event handling, reaction window management and event scheduling. While conforming to the TROM operational semantics, event handling includes updating the history of the events in the simulation scenario. The objects composing the functional model of the simulator include an event handler, a reaction window manager, an event scheduler, and a time manager. When the event handler processes a computational step according to the transition fired, the reaction window manager updates the reaction vector in the status of the TROM object, and requests the event scheduler to schedule or unschedule events as required. These operations have been included in the formal specifications, allowing an easy derivation of an implementation.

4.4 Conformance to TROM Operational Semantics

The simulator as specified in VDM conforms to the operational semantics of TROM given in Section 2.3.2. We establish this claim by showing that the "handle-transition" operation is consistent with the semantics of a computational step as defined in [Ach95]. A similar approach can be followed to show consistency of other operations and functions in the formal specifications.

The "handle-transition" operation, which specifies a computational step, is most fundamental in the functional model of the simulator, and its formal specification is shown in Figure 16. The handle-transition operation *fires* outstanding reactions, *disables* reactions associated with a time constraint when the TROM enters a disabling state, and *enables* reactions time-constrained by a transition. We show that the "handle-transition" operation is consistent with the operational semantics by first explaining the implementation-dependent operations which update the history of the computational step. We then focus on the statements which specify the firing, disabling, and enabling of reactions. The operations "update-history-fire-reaction", "update-history-disable-reaction", and "update-history-enable-reaction" update the fields of the simulation event being handled, so that the outcome of the transition is saved. These operation do not affect the status of the TROM objects in the system. The other operations which are used by the "handle-transition" operation include "fire-reaction", "disable-reaction", and "enable-reaction".

Each time constraint has an associated reaction subvector, which includes the outstanding reaction windows corresponding to the constrained event. To fire a reaction, we first access the reaction subvector associated with the time constraint which constrains the event being handled. If the occurrence time of the simulation event being handled is within a reaction window from the reaction subvector, the "fire-reaction" operation removes the reaction window from that reaction subvector. When the TROM object enters one of the disabling states for a time constraint, the "disable-reaction" operation removes all the reaction windows from the reaction subvector associated with that time constraint. Finally, when the transition being handled is associated with a time constraint, the "enable-reaction" operation adds a reaction window to the reaction subvector of that transition. This conforms to the notion of computational step as defined in the operational semantics. We conclude by saying that the "handle-transition" operation is consistent with the operational semantics of TROM.

Chapter 5

Formal Verification Methodology

When a system is developed it is desirable to demonstrate that the system satisfies safety, liveness, and time-bounded properties. Proving that such properties are satisfied by the system is crucial in the development of dependable safety-critical systems. There is no known formal verification technique for reactive systems designed on object-oriented principles. Our effort is probably the first attempt. The formal methodology we introduce applies to verification of systems described in the TROM *formalism. A future goal is to mechanize the verification process using the PVS specification and verification system as a back-end.*

5.1 Introduction

This chapter includes a review of formal verification techniques for real-time systems. Section 5.2 describes the steps in verifying safety and liveness properties, and how the verification process can be mechanized using an automated reasoning system. It includes a brief introduction to PVS, a prototype verification system which can be used for specification and verification of real-time systems. It also describes how real-time behavior can be formalized with a computational model. Section 5.3 describes the notion of duration used in specifying properties, and axioms. It includes an outline of the semantics of the since operator introduced by Shankar [Sha93c], and the relationship between the since operator and absolute times. Section 5.4 introduces our methodology for formal verification of TROM-based systems, detailing the steps in proving safety properties.

5.2 Verification of Real-Time Systems

Real-Time systems use time to ensure that tasks are scheduled in a timely manner, deadlines are met, processes are synchronized, and race conditions are avoided [Sha93c]. In addition, systems in safety-critical contexts have to satisfy crucial requirements. Verification implies proving that these requirements are satisfied.

5.2.1 Verifying Safety and Liveness Properties

Real-time systems used in safety-critical contexts have to adhere to strict safety and liveness properties. A safety property represents an assertion that something bad will never happen, while a liveness property represents an assertion that something good will eventually happen. Such properties can only be established by a rigorous analysis of the system under development, using sound mathematical proofs. However, this exercise can be very expensive in terms of effort and time.

To demonstrate that such requirements are met, the real-time behavior of such systems has to be formalized. A computational model describing the attributes of a reactive system is needed to support formal verification. Several approaches have been proposed for formal verification of real-time systems. Among these, SIP [FS93] uses the reasoning system FRAPPE to deduce the properties of a real-time reactive system through simulation. Ostroff [Ost94] proposes TTM/RTTL as a framework for specifying and verifying real-time reactive systems. The basis of TTM/RTTL is a reachability analysis of the possible states of the system to analyze its behavior.

5.2.2 Mechanized Verification Using an Automated Reasoning System

Mechanical assistance in the form of a theorem prover has been proposed for formal verification. An automated reasoning system needs to provide both an expressive logic and powerful automation to be able to support mechanical verification. The specification system should provide a sound logic allowing clear and abstract specifications. The strategies provided should be sound and readable for difficult theorems. A verification system can detect flaws in the design at an early stage, thus reducing the costs of remedying faulty systems.

PVS [ORS92] is a specification and reasoning system, which allows the mechanical checking of specifications. It supports an expressive logic with a powerful proof checker. We further discuss PVS in the next section, and show how our methodology is influenced by the computational model used for verification in PVS.

5.2.3 PVS - A Prototype Verification System

PVS [ORS92] consists of a specification language based on higher-order logic, and an interactive proof checker that uses powerful arithmetic decision procedures. It includes a parser, a pretty-printer, and a type-checker, allowing the development of specifications in a concise and consistent manner. The logic of PVS is strongly typed, with a rich type system. The specifications are written as parameterized theories, with constraints attached to the parameters. The types in a specification also can have constraints attached to them. The language allows the definition of abstract data types, predicate subtypes, and dependent types. The proof checker supports the efficient development of proofs. It implements a set of powerful primitive inference rules, and a mechanism for composing these rules into proof strategies. This is useful in composing frequently used patterns of rules into a single step. The PVS system also allows rerunning proofs.

5.2.4 Formalizing Real-Time Behavior with a Computational Model

In Shankar's computational model [Sha93b], a *state* is a mapping of program variables to values, a *trace* is an infinite sequence of states, and a *program variable* maps a given state to the value of the variable in that state. *Time* is a special program variable whose value is not modified by a program. A *behavior* is a trace where the value of Time is non-decreasing and eventually increases above any bound. A *rooted behavior* is a behavior where the initial value of Time is 0. A program identifies a set of rooted behaviors; a specification also identifies a set of behaviors. A program satisfies a specification if the set of behaviors given by the program is a subset of the behaviors identified by the specification.

A *state predicate* is a predicate on states; an *atomic action* is a binary relation between states. A program is described in terms of an initialization state predicate and a sequence of atomic actions. In any behavior satisfying a given program, the initial state must satisfy the initialization predicate, and each pair of adjacent states must satisfy one of the atomic actions of the program. An *invariance assertion* is a state predicate which is invariant over a behavior; that is, it holds of each state in the behavior. It is typical to use induction over the states of an arbitrary behavior satisfying a program, to show that the program satisfies an invariant. Time-bounded versions of certain liveness properties can be expressed as invariance assertions.

- An *initialization assertion* has the form **initially**$\{P\}$, where the state predicate P represents the initial state of the system.
- An *invariant assertion* on the state predicate P is stated as **invariant**$\{P\}$.

To prove that **invariant**$\{P\}$ holds of a program with initialization predicate *init* and atomic actions S_i, we show that *init* $\supset P$ and that the Hoare assertion $\{P\}S_i\{P\}$ holds for each atomic action S_i.

5.3 Notion of Duration

We use the since operator in deriving the time-constraint and synchronization axioms, and in specifying invariance assertions for the safety property. We also use atomic actions to specify the program representing the subsystem. These axioms and actions are used in the proof of the invariance assertions to establish the correctness of programs that exhibit real-time behavior.

5.3.1 Semantics of since Operator

The value of since for a given predicate at a given state in a program execution, is the time that has elapsed since the predicate last held. It is similar to the *punch* operator introduced by Carruth and Misra [CM92] as an extension to Chandy and Misra's Unity logic [CM88]; *punch* operates on assertions and records the absolute time at which the assertion last went from being false to true. since is also similar to the *duration* operator introduced by Maler, Manna, and Pnueli [MMP91]; the value of *duration* for a given formula at any state s is the largest time duration ending in s for which the formula has continuously held.

Additional axioms are needed to capture real-time behavior.

- The initial value of $\mid P \mid$ for any state predicate P is 1.

initially$\{|\ P\ | = 1\}$, for all P. **(init)**

- If P is true in the precondition of an atomic action, then the value of $|\ P\ |$ in the postcondition is equal to the *delay* for the action.

  $\{r = Time \wedge P\}\ S\ \{|\ P\ | = Time - r\}$, for all P.

- If P is false in the precondition of an action, then the value of $|\ P\ |$ in the postcondition is got by adding the *delay* for the action to the precondition value of $|\ P\ |$.

  $\{r = Time \wedge t = |\ P\ | \wedge \neg P\}\ S\ \ \{|\ P\ | = t + (Time - r)\}$, for all P.

5.3.2 Relationship between since and Absolute Times

State Predicate	*Absolute time at which the predicate becomes false*
P	t_P
Q	t_Q
$P \wedge Q$	$t_{P \wedge Q}$
$P \vee Q$	$t_{P \vee Q}$
$P \wedge \neg Q$	$t_{P \wedge \neg Q}$
$\|\ P\ \| \leq x$	$t_{\|P\| \leq x}$
$\|\ (\|\ P\ \| \leq x)\ \| \leq y$	$t_{\|(\|P\| \leq x)\| \leq y}$

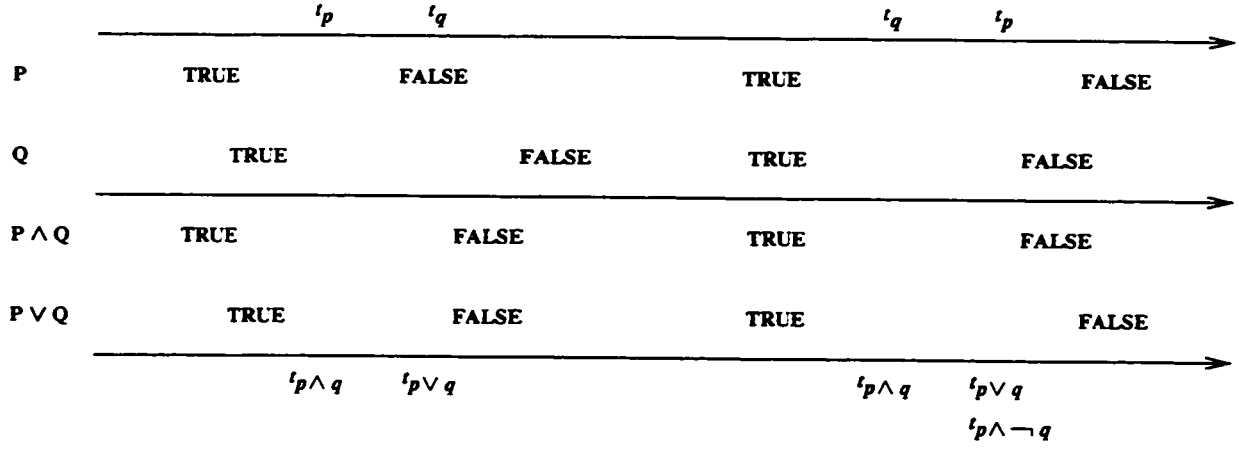


Figure 17: Absolute times at which a predicate becomes *false*.

In our attempt to prove the lemmas representing the invariance assertions, we transform the formulas that use the since operator for the specification of durations in state predicates into inequalities over real-time variables. The time-constraint and synchronization axioms, and the lemmas are rewritten in terms of *absolute times*. Any predicate formula of the form

$$P \supset since(Q) < x$$

where P and Q are state predicates, can be transformed to the linear inequality

$$t - t_Q < x$$

where predicate P is *true* at time t, and t_Q is the absolute time prior to t when predicate Q was last *true*. For a predicate formula of the form

$$P \supset since(Q) - since(R) < x$$

we get by similar reasoning the inequality

$$t_R - t_Q < x$$

for all values of t such that predicate P is *true* at time t, and t_Q and t_R are respectively the absolute times prior to t when predicates Q and R were last *true*. Similar inequalities can be derived for other boolean operators. All formulas in the invariance assertions are of these two kinds.

Shankar [Sha93b] introduces several lemmas regarding invariants and the behavior of the **since** operator. We show the relationship between the **since** operator and absolute times in the invariants that follow. We rewrite the invariants over the **since** operator in terms of the absolute times at which the predicates become *false*. The variables P and Q in the statements below range over state predicates, as represented in Figure 17.

- **INV1.1 :** $\{|(|\ P\ | \leq x)| \leq y \supset |\ P\ | \leq x + y\}$
  - **INV2.1 :** $t_{|(|P| \leq x)| \leq y} = t_{|P| \leq x} + y = t_{|P|} + x + y$
- **INV1.2 :** $\{|\ P \vee Q\ | \leq \min(|\ P\ |, |\ Q\ |)\}$
  - **INV2.2 :** $t_{P \vee Q} = \max(t_P, t_Q)$
- **INV1.3 :** $\{|\ P \vee Q\ | = |\ P\ | \vee |\ P \vee Q\ | = |\ Q\ |\}$
  - **INV2.3 :** $t_{P \vee Q} = t_P \vee t_{P \vee Q} = t_Q$
- **INV1.4 :** $\{|\ P \wedge Q\ | = |\ P\ | \vee |\ P \wedge \neg Q\ | = |\ P\ |\}$
  - **INV2.4 :** $t_{P \wedge Q} = t_P \vee t_{P \wedge \neg Q} = t_P$
- **INV1.5 :** $\{\max(|\ P\ |, |\ Q\ |) \leq |\ P \wedge Q\ |\}$
  - **INV2.5 :** $t_{P \wedge Q} = \min(t_P, t_Q)$
- **INV1.6 :** $\{|\ P\ | \leq |\ P \wedge Q\ |\}$

- **INV2.6 :** $t_P \geq t_{P \wedge Q}$

- **INV1.7 :** $\{|\ Q\ | \leq |\ P \wedge Q\ |\}$

  - **INV2.7 :** $t_Q \geq t_{P \wedge Q}$

5.4 A Methodology for Formal Verification

Our approach to verification is to prove that a program based on the specifications of TROM classes and subsystems satisfies the safety properties for the correct functioning of the system. We derive axioms from the timing constraints and from the synchronization messages specified as external events in the TROM class specifications. We represent transition specifications as atomic actions, that is, binary relations on system states, taking into consideration the enabling condition, the port-condition, and the post-condition for the transition. We specify the property to be verified as an invariance assertion for a state predicate; the state predicate is sometimes rewritten as a conjunction of simpler state predicates to facilitate the proof. We then attempt to prove the lemmas representing the invariance assertions. In proving an invariance assertion, we rewrite the formulas having state predicates involving durations as inequalities over real-time variables.

In verifying a TROM subsystem, we consider the state of the system as the *statuses* of all TROMs in the subsystem, including the current state of each TROM, its assignment vector, and its reaction vector. For each TROM class, we define a higher-order function which takes a *system state* and a TROM object, and returns the *status* of the TROM object in that state of the system. We derive *atomic actions* to represent the transition specifications of the TROM class, taking into consideration the synchronization of messages. We also derive axioms based on the timing constraints of the TROM class and on the synchronized messages.

5.4.1 Deriving Axioms from Class Specifications

Each possible transition in the state of the system is represented by an atomic action, that is, a binary relation on two system states. A transition specification representing an *internal* event is translated into an atomic action describing a change of status for only one TROM object; the statuses of all other TROMs in the system remain unchanged. When considering transition specifications representing *external* events, an atomic action describes a change of status for a TROM sending the message and for a TROM receiving the message. This may lead to several actions representing an event, since the TROM sending the message as an output event may do so in several different statuses, and the receiving

TROM may accept the message as an input event in several different statuses, as represented by different transition specifications in the TROM class specification. The *port-condition*, *enabling-condition*, and *post-condition* of the transition specifications are included as *boolean* expressions in the action. The post-conditions will embody any change in the *assignment vector* of the TROMs involved in the action. The *reaction vector* of each TROM is updated in the action, according to the TROM operational semantics.

We use the **since** operator to express durations in state predicates when translating timing constraints and synchronization messages into axioms. The **since** operator is also used to specify durations for properties stated as *bounded-time* properties. We also use time charts representing interactions between TROM objects in deriving time-constraint and synchronization axioms. The time charts give a graphical representation of the transitions and the synchronization messages, together with the corresponding timing constraints.

We can derive time-constraint axioms from the specifications of the timing constraints as follows: An upper time bound on the firing of a transition is represented as a constraint that if the TROM is in the source state of the transition, then the time elapsed since the event triggering the transition was enabled is less than the upper time bound. A lower time bound on the firing of a transition is represented as a constraint: if the TROM is in the destination state of the transition, then the time elapsed since the event triggering the transition was enabled is greater than the lower time bound. Other axioms can be included to capture the same properties in other states of the TROM prior to entering the source state or after exiting the destination state of the particular transition, based on the duration since a specific occurrence of an event. Alternatively, the *constrained-event* axiom (*11:ce*) and the transition axiom (*9:TR*) from the logical semantics (see Appendix B) can be used. Assume that for a TROM $\mathcal{A}$, an event f triggers the event e according to the *constrained-event* axiom (*11:ce*)

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

Since every event is associated with a transition, and the transition axiom (*9:TR*) specifies the source and destination states for the transition, we can derive from axiom (*9:TR*) of Appendix B the following two axioms:

$$Hold(s_3, t) \wedge Occur(e, p_i, t) \wedge Meet(t, t') \rightarrow Hold(s_4, t')$$

$$Hold(s_1, t_a) \wedge Occur(f, p_j, t_a) \wedge Meet(t_a, t'_a) \rightarrow Hold(s_2, t'_a)$$

In writing these axioms we have assumed that the post-conditions for the transitions hold. If $s_2 = s_3$, then we reason that when the TROM $\mathcal{A}$ is in state s_4, the duration between the

instances when TROM $\mathcal{A}$ was at s_1 and s_2 lies in the interval $[l, u]$. Consequently, we write the axioms using since:

$$A = s_4 \supset since(A = s_1) - since(A = s_2) > l$$

$$A = s_4 \supset since(A = s_1) - since(A = s_2) < u$$

If $s_2 \neq s_3$, then for each state between s_2 and s_3 (as described in history), we write the above axioms. In addition, the above axioms remain valid for every state following s_4 in the history of A. Some of these axioms are redundant, as we shall see in the next section.

We derive synchronization axioms from the transition specifications for external events, and synchronized messages. For each synchronized message, the two TROMs sharing the synchronized action enter a particular status at the same time, and the duration since the occurrence of that event will be the same for both TROMs as long as they both remain in the respective statuses. In addition, this assertion will hold as long as there is no other synchronized message between the two TROMs. This fact can be expressed by additional axioms. These axioms can be written from the *synchrony* axiom (SY), and the *transition* axiom (*9:TR*). For each external event, we initialize the *synchrony* axiom, and for each pair of synchronized TROMs we instantiate the *transition* axiom from the logical semantics, for both TROMs.

As described above, the axioms generated from the class specifications conform to the logical semantics of TROM. Consequently, the axiomatization basis for our verification methodology is sound.

5.4.2 Time Charts and Lemmas

A *time chart* specifies the status of the system at various time points measured absolutely, during the simulation process. The horizontal axis represents absolute times from 0 to infinity. In the vertical axis, we show the different TROM objects in the system configuration. We can thus show the status of the system at significant time points. A coordinate in the time chart represents the status of a TROM object at that point. The status of the system at any time t is obtained by taking all the coordinates on a vertical line drawn through the point t on the horizontal axis.

If b_i and b_j denote two vertical lines through the absolute times i, and j, where $j > i$, such that the status of a TROM object at time i is the same as the status of the TROM object for all time points k, where $i \leq k < j$, then any property of the TROM object that was true during this duration can be expressed using since as well as absolute times.

For example, if a predicate P is true at time $t > i$, and P implies that another predicate Q was last true at time i, then we can express this assertion as

$$P \supset since(Q) < x, \ where \ x = j - i.$$

5.4.3 Proving Properties

We specify the property to be verified as a predicate on the state of the system, that is, the statuses of the TROMs in the system. The predicate can be rewritten as a conjunction of simpler predicates. For each predicate, we include a lemma to verify the invariance assertion for that predicate on an arbitrary behavior of the system. In the proof steps for the lemma, we translate the formulas which use the **since** operator into inequalities over real-time variables. A program representing an arbitrary behavior of the system will satisfy the invariance assertions representing the property predicate. We also introduce other lemmas representing invariance assertions on the program to simplify the proof steps.

The axioms and lemmas are expressed as linear inequalities involving variables which denote the significant instances of changes in statuses of the TROM objects. The property to be proved is expressed as another linear inequality. Thus, we restate a lemma representing a property as a linear inequality over the same variables. We then have to prove a linear inequality from a set of given linear inequalities representing axioms.

Chapter 6

Verification: A Case Study

We illustrate the formal verification methodology with a case study. We use the railroad crossing example to show the steps in proving a safety property. We introduce time charts to indicate how the axioms and lemmas can be written as inequalities with variables on absolute times.

6.1 Introduction

This chapter shows how the methodology can be applied to prove a safety property for the Train-Gate-Controller system. Section 6.2 describes the transition actions, the time-constraint axioms, and the synchronization axioms for a simple version of the Train-Gate-Controller system. Section 6.3 describes how the safety property can be expressed as invariance assertions on the state of the system. Section 6.4 shows the steps in proving the lemmas representing the safety property. We describe how time charts are used to derive the axioms, and translate formulas involving the **since** operator into inequalities over absolute times, in specifying invariance assertions. We include the atomic actions and the axioms derived, and show the proof steps for the lemmas. Section 6.5 concludes the chapter with a detailed description of the proof for a generalized version of the Train-Gate-Controller system.

6.2 Axioms

In this section we show how the axioms derived from the TROM specifications can be used in the formal verification of a simple version of the Train-Gate-Controller system. The system we describe consists of one train, one controller, and one gate. The safety requirement is that whenever the train is in the crossing, the gate is closed and the controller is monitoring. Since we have only one train, one controller, and one gate, the assignment vector and the reaction vector of the TROM objects become irrelevant in specifying safety properties; they are omitted in the specification of the actions. Since each TROM class is instantiated only once, we do not need higher-order functions to access the status of the objects. In the configuration of the subsystem, we have one port-link between the train and the controller, and one port-link between the controller and the gate. Figure 18 shows the state transition diagram for the TROMs in the system. Figure 19 shows the time chart, including all the possible transitions, the synchronization messages, and the timing constraints.

6.2.1 Transition Actions, Time Constraint and Synchronization Axioms

Transition Actions

- **ACT3.1** : Train approaches crossing and sends **Near** message to controller.
  - $\lambda\ s_0, s_1 : train(s_0) = idle \wedge train(s_1) = toCross \wedge$
    $controller(s_0) = idle \wedge controller(s_1) = activate \wedge$
    $gate(s_1) = gate(s_0)$

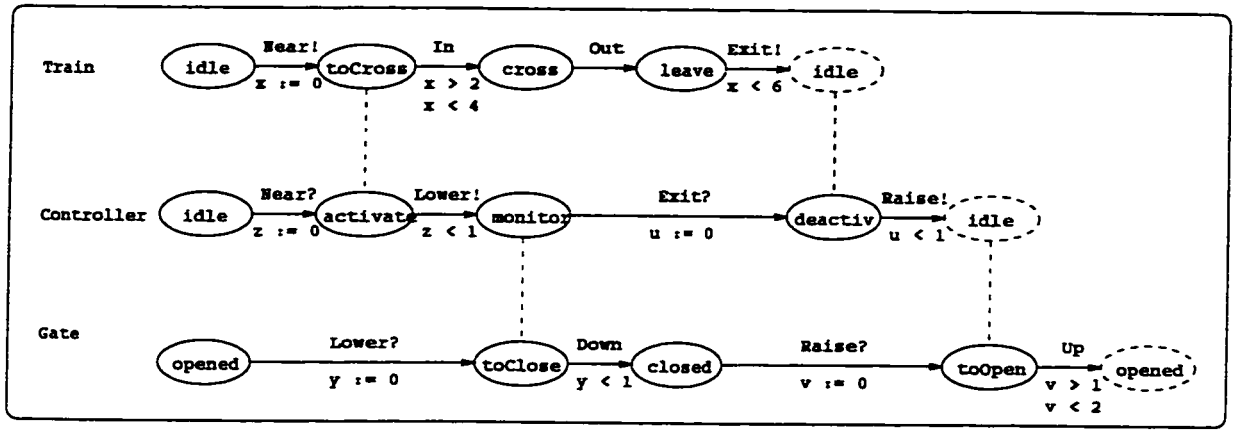


Figure 18: State transition diagram for Train-Gate-Controller system.

- **ACT3.2** : Train enters crossing - internal event In.
  - $\lambda\ s_0, s_1 : train(s_0) = toCross \wedge train(s_1) = cross \wedge$
    $controller(s_1) = controller(s_0) \wedge gate(s_1) = gate(s_0)$
- **ACT3.3** : Train leaves crossing - internal event Out.
  - $\lambda\ s_0, s_1 : train(s_0) = cross \wedge train(s_1) = leave \wedge$
    $controller(s_1) = controller(s_0) \wedge gate(s_1) = gate(s_0)$
- **ACT3.4** : Train sends Exit message to controller and becomes idle.
  - $\lambda\ s_0, s_1 : train(s_0) = leave \wedge train(s_1) = idle \wedge$
    $controller(s_0) = monitor \wedge controller(s_1) = deactivate \wedge$
    $gate(s_1) = gate(s_0)$
- **ACT3.5** : Controller sends Lower message to gate and starts to monitor crossing.
  - $\lambda\ s_0, s_1 : controller(s_0) = activate \wedge controller(s_1) = monitor \wedge$
    $gate(s_0) = opened \wedge gate(s_1) = toClose \wedge$
    $train(s_1) = train(s_0)$
- **ACT3.6** : Controller sends Raise message to gate and becomes idle.
  - $\lambda\ s_0, s_1 : controller(s_0) = deactivate \wedge controller(s_1) = idle \wedge$
    $gate(s_0) = closed \wedge gate(s_1) = toOpen \wedge$
    $train(s_1) = train(s_0)$
- **ACT3.7** : Gate closes - internal event Down.

- $\lambda\ s_0, s_1 : gate(s_0) = toClose \wedge gate(s_1) = closed \wedge$
  $controller(s_1) = controller(s_0) \wedge train(s_1) = train(s_0)$

- **ACT3.8** : Gate opens - internal event Up.

  - $\lambda\ s_0, s_1 : gate(s_0) = toOpen \wedge gate(s_1) = opened \wedge$
    $controller(s_1) = controller(s_0) \wedge train(s_1) = train(s_0)$

Time Constraint Axioms

- **TC_AX4.1** : Train starts to cross (internal event In) within window of 2 to 4 time units after sending Near message to controller.

  - **TC_AX4.1.1** : $\lambda\ s : train(s) = cross \supset$
    $since(\lambda\ s_0 : train(s_0) = idle)(s) > 2$
  - **TC_AX4.1.2** : $\lambda\ s : train(s) = cross \supset$
    $since(\lambda\ s_0 : train(s_0) = idle)(s) -$
    $since(\lambda\ s_1 : train(s_1) = toCross)(s) > 2$
  - **TC_AX4.1.3** : $\lambda\ s : train(s) = leave \supset$
    $since(\lambda\ s_0 : train(s_0) = idle)(s) -$
    $since(\lambda\ s_1 : train(s_1) = toCross)(s) > 2$
  - **TC_AX4.1.4** : $\lambda\ s : train(s) = toCross \supset$
    $since(\lambda\ s_0 : train(s_0) = idle)(s) < 4$
  - **TC_AX4.1.5** : $\lambda\ s : train(s) = cross \supset$
    $since(\lambda\ s_0 : train(s_0) = idle)(s) -$
    $since(\lambda\ s_1 : train(s_1) = toCross)(s) < 4$
  - **TC_AX4.1.6** : $\lambda\ s : train(s) = leave \supset$
    $since(\lambda\ s_0 : train(s_0) = idle)(s) -$
    $since(\lambda\ s_1 : train(s_1) = toCross)(s) < 4$

- **TC_AX4.2** : Train sends Exit message to controller within 6 time units after sending Near message to controller.

  - **TC_AX4.2.1** : $\lambda\ s : train(s) = leave \supset$
    $since(\lambda\ s_0 : train(s_0) = idle)(s) < 6$

- **TC_AX4.3** : Controller sends Lower message to gate within 1 time unit after receiving Near message from train.

- **TC_AX4.3.1** : $\lambda\ s : controller(s) = activate \supset$
  $since(\lambda\ s_0 : controller(s_0) = idle)(s) < 1$
- **TC_AX4.3.2** : $\lambda\ s : controller(s) = monitor \supset$
  $since(\lambda\ s_0 : controller(s_0) = idle)(s) -$
  $since(\lambda\ s_1 : controller(s_1) = activate)(s) < 1$
- **TC_AX4.3.3** : $\lambda\ s : controller(s) = deactivate \supset$
  $since(\lambda\ s_0 : controller(s_0) = idle)(s) -$
  $since(\lambda\ s_1 : controller(s_1) = activate)(s) < 1$

- **TC_AX4.4** : Controller sends Raise message to gate within 1 time unit after receiving Exit message from train.
  - **TC_AX4.4.1** : $\lambda\ s : controller(s) = deactivate \supset$
    $since(\lambda\ s_0 : controller(s_0) = monitor)(s) < 1$
  - **TC_AX4.4.2** : $\lambda\ s : controller(s) = idle \supset$
    $since(\lambda\ s_0 : controller(s_0) = monitor)(s) -$
    $since(\lambda\ s_1 : controller(s_1) = deactivate)(s) < 1$
  - **TC_AX4.4.3** : $\lambda\ s : controller(s) = activate \supset$
    $since(\lambda\ s_0 : controller(s_0) = monitor)(s) -$
    $since(\lambda\ s_1 : controller(s_1) = deactivate)(s) < 1$

- **TC_AX4.5** : Gate closes (internal event Down) within 1 time unit after receiving Lower message from controller.
  - **TC_AX4.5.1** : $\lambda\ s : gate(s) = toClose \supset$
    $since(\lambda\ s_0 : gate(s_0) = opened)(s) < 1$
  - **TC_AX4.5.2** : $\lambda\ s : gate(s) = closed \supset$
    $since(\lambda\ s_0 : gate(s_0) = opened)(s) -$
    $since(\lambda\ s_1 : gate(s_1) = toClose)(s) < 1$
  - **TC_AX4.5.3** : $\lambda\ s : gate(s) = toOpen \supset$
    $since(\lambda\ s_0 : gate(s_0) = opened)(s) -$
    $since(\lambda\ s_1 : gate(s_1) = toClose)(s) < 1$

- **TC_AX4.6** : Gate opens (internal event Up) within window of 1 to 2 time units after receiving Raise message from controller.
  - **TC_AX4.6.1** : $\lambda\ s : gate(s) = opened \supset$
    $since(\lambda\ s_0 : gate(s_0) = closed)(s) > 1$

- **TC_AX4.6.2** : $\lambda\ s : gate(s) = opened \supset$
$since(\lambda\ s_0 : gate(s_0) = closed)(s) -$
$since(\lambda\ s_1 : gate(s_1) = toOpen)(s) > 1$
- **TC_AX4.6.3** : $\lambda\ s : gate(s) = toClose \supset$
$since(\lambda\ s_0 : gate(s_0) = closed)(s) -$
$since(\lambda\ s_1 : gate(s_1) = toOpen)(s) > 1$
- **TC_AX4.6.4** : $\lambda\ s : gate(s) = toOpen \supset$
$since(\lambda\ s_0 : gate(s_0) = closed)(s) < 2$
- **TC_AX4.6.5** : $\lambda\ s : gate(s) = opened \supset$
$since(\lambda\ s_0 : gate(s_0) = closed)(s) -$
$since(\lambda\ s_1 : gate(s_1) = toOpen)(s) < 2$
- **TC_AX4.6.6** : $\lambda\ s : gate(s) = toClose \supset$
$since(\lambda\ s_0 : gate(s_0) = closed)(s) -$
$since(\lambda\ s_1 : gate(s_1) = toOpen)(s) < 2$

Synchronization Axioms

1. **SYN_AX5.1** : Near message from train to controller
   - States entered concurrently: train = toCross, controller = activate
   - **Axioms** :
     - **SYN_AX5.1.1** : $\lambda\ s : train(s) = toCross \wedge controller(s) = activate \supset$
$since(\lambda\ s_0 : train(s_0) = idle)(s) =$
$since(\lambda\ s_1 : controller(s_1) = idle)(s)$
     - **SYN_AX5.1.2** : $\lambda\ s : train(s) = toCross \wedge controller(s) = monitor \supset$
$since(\lambda\ s_0 : train(s_0) = idle)(s) =$
$since(\lambda\ s_1 : controller(s_1) = idle)(s)$
     - **SYN_AX5.1.3** : $\lambda\ s : train(s) = cross \wedge controller(s) = monitor \supset$
$since(\lambda\ s_0 : train(s_0) = idle)(s) =$
$since(\lambda\ s_1 : controller(s_1) = idle)(s)$
     - **SYN_AX5.1.4** : $\lambda\ s : train(s) = leave \wedge controller(s) = monitor \supset$
$since(\lambda\ s_0 : train(s_0) = idle)(s) =$
$since(\lambda\ s_1 : controller(s_1) = idle)(s)$
2. **SYN_AX5.2** : Exit message from train to controller
   - States entered concurrently: train = idle, controller = deactivate

- **Axiom :**
  - **SYN_AX5.2.1** : $\lambda\ s : train(s) = idle \wedge controller(s) = deactivate \supset since(\lambda\ s_0 : train(s_0) = leave)(s) = since(\lambda\ s_1 : controller(s_1) = monitor)(s)$

3. **SYN_AX5.3** : Lower message from controller to gate
   - States entered concurrently: controller = **monitor**, gate = **toClose**
   - **Axioms :**
     - **SYN_AX5.3.1** : $\lambda\ s : controller(s) = monitor \wedge gate(s) = toClose \supset since(\lambda\ s_0 : controller(s_0) = activate)(s) = since(\lambda\ s_1 : gate(s_1) = opened)(s)$
     - **SYN_AX5.3.2** : $\lambda\ s : controller(s) = monitor \wedge gate(s) = closed \supset since(\lambda\ s_0 : controller(s_0) = activate)(s) = since(\lambda\ s_1 : gate(s_1) = opened)(s)$
     - **SYN_AX5.3.3** : $\lambda\ s : controller(s) = deactivate \wedge gate(s) = closed \supset since(\lambda\ s_0 : controller(s_0) = activate)(s) = since(\lambda\ s_1 : gate(s_1) = opened)(s)$

4. **SYN_AX5.4** : Raise message from controller to gate
   - States entered concurrently: controller = **idle**, gate = **toOpen**
   - **Axiom :**
     - **SYN_AX5.4.1** : $\lambda\ s : controller(s) = idle \wedge gate(s) = toOpen \supset since(\lambda\ s_0 : controller(s_0) = deactivate)(s) = since(\lambda\ s_1 : gate(s_1) = closed)(s)$

6.2.2 Axioms in Absolute Times

We use the following notation in expressing the axioms in absolute times. The subscript i in a variable for absolute time indicates that the gate is being **closed** for the *i-th* time. As illustrated in Figure 19, all events occurring when the gate is being closed for the *i-th* time are subscripted with i. Table 2 gives the absolute times at which an event occurs in the simple version of the Train-Gate-Controller system.

We illustrate this with the following example. The axioms given below are rewritten using absolute times.

Event	Absolute Time	Description
Near	a_i	Message from train to controller
Lower	b_i	Message from controller to gate
Down	c_i	Internal to gate
In	d_i	Internal to train
Out	e_i	Internal to train
Exit	f_i	Message from train to controller
Raise	g_i	Message from controller to gate
Up	h_i	Internal to gate

Table 2: Absolute times at which the events occur.

$$\textbf{TC\_AX4.1.2} : \lambda\ s : train(s) = cross \supset since(\lambda\ s_0 : train(s_0) = idle)(s) - since(\lambda\ s_1 : train(s_1) = toCross)(s) > 2$$

$$\textbf{TC\_AX4.1.5} : \lambda\ s : train(s) = cross \supset since(\lambda\ s_0 : train(s_0) = idle)(s) - since(\lambda\ s_1 : train(s_1) = toCross)(s) < 4$$

$$\textbf{TC\_AX4.1.2} : d_i - a_i > 2,\ \forall x : d_i < x < e_i$$

$$\textbf{TC\_AX4.1.5} : d_i - a_i < 4,\ \forall x : d_i < x < e_i$$

6.3 Safety Property

We specify the safety property for the simplified version of the Train-Gate-Controller system, and give the state predicates whose conjunction is equivalent to the safety predicate. An invariance assertion is used for each predicate; we include the proof steps for the lemmas representing the invariance assertions.

- Safety Property:
  - *When the train is crossing, the gate is closed and the controller is monitoring.*
- Invariance Assertions:
  1. The gate is already closed when the train starts to cross.
  2. The gate remains closed while the train is crossing.
  3. The controller is already monitoring when the train starts to cross.

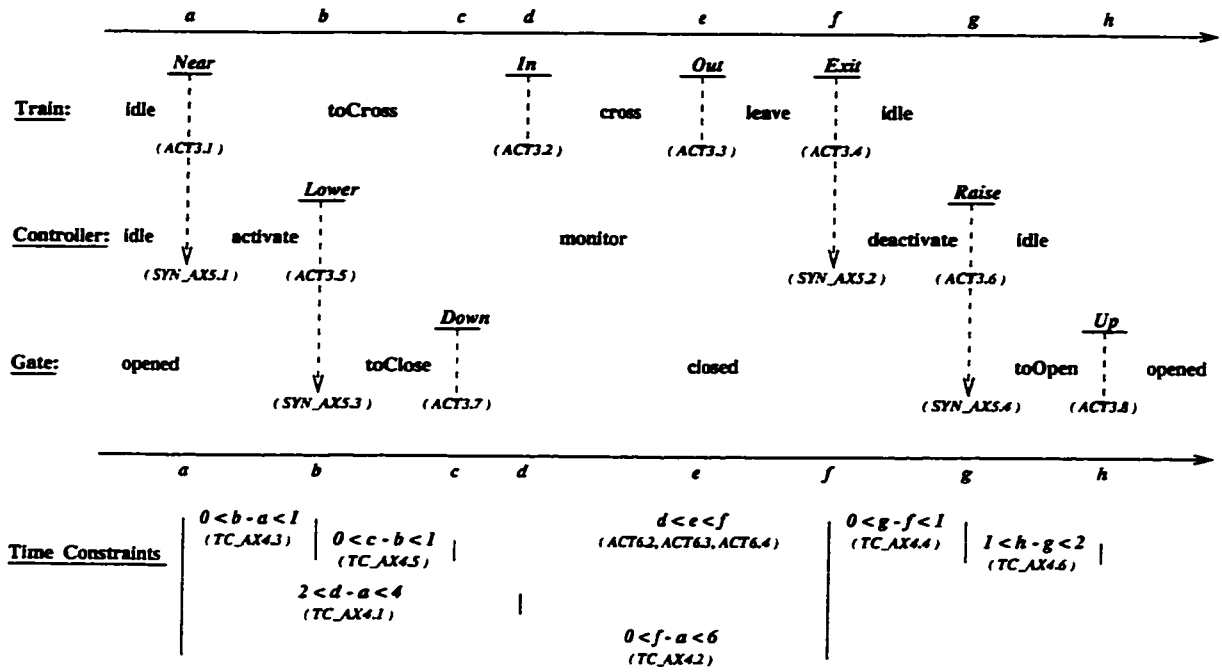


Figure 19: Time chart for Train-Gate-Controller system.

4. The controller remains monitoring while the train is crossing.
5. The controller is not deactivated before the train leaves the crossing.
6. The gate is not raised before the controller is deactivated.
7. The gate is not raised before the train leaves the crossing.

6.4 Proof Steps

Safety Property as Invariance Assertions

- Proof Steps for the Invariance Assertions:
  - **LEMMA6.1** : The gate is already closed when the train starts to cross.
    - **Lemma** : $\lambda\ s : train(s) = cross \supset$
      $since(\lambda\ s_0 : gate(s_0) = toClose)(s) >$
      $since(\lambda\ s_1 : train(s_1) = toCross)(s)$
      $\Longrightarrow (x - c_i) > (x - d_i),\ \forall x : d_i < x < e_i$
      $\Longrightarrow c_i < d_i$
    - **Proof** :
      1. **TC_AX4.1.2** : $d_i - a_i > 2$

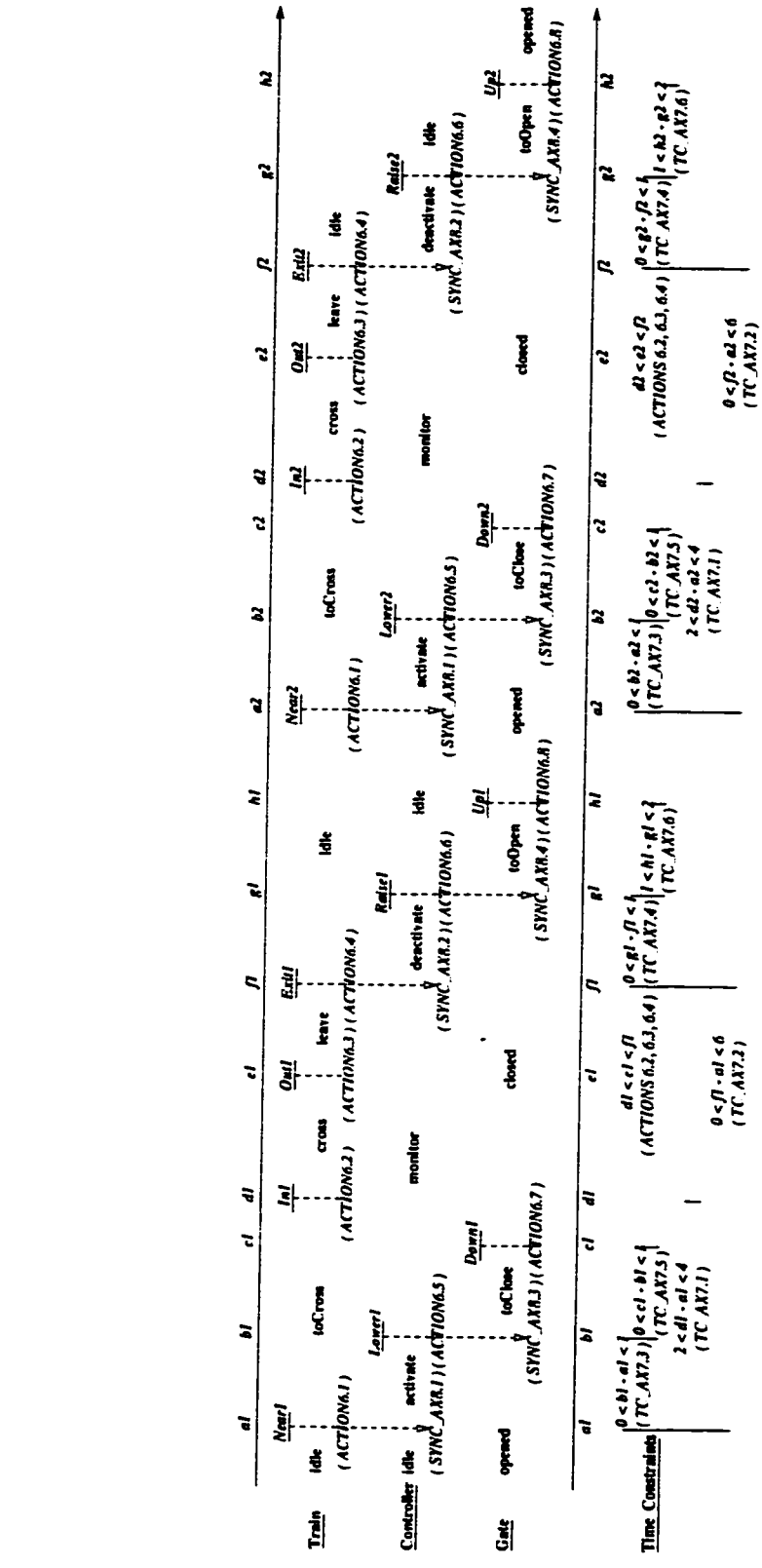


Figure 20: Time chart for Train-Gate-Controller system, showing wrap-around.

2. From (1) : $d_i - 2 > a_i$

3. **TC_AX4.3.2** : $b_i - a_i < 1$

4. From (3) : $b_i - 1 < a_i$

5. From (2) and (4) : $b_i - 1 < a_i < d_i - 2$

6. From (5) : $b_i < d_i - 1$

7. **TC_AX4.5.2** : $c_i - b_i < 1$

8. From (7) : $c_i < b_i + 1$

9. From (3) : $b_i + 1 < a_i + 2$

10. From (8) and (9) : $c_i < b_i + 1 < a_i + 2$

11. From (10) : $c_i < a_i + 2$

12. From (5) : $a_i + 2 < d_i$

13. From (11) and (12) : $c_i < a_i + 2 < d_i$

14. From (13) : $c_i < d_i$

- **LEMMA6.2** : The gate remains closed while the train is crossing.
  - **Lemma** : $\lambda\ s : train(s) = leave \supset$
    $since(\lambda\ s_0 : gate(s_0) = toClose)(s) >$
    $since(\lambda\ s_1 : train(s_1) = toCross)(s)$
    $\Longrightarrow (x - c_i) > (x - d_i),\ \forall x : e_i < x < f_i$
    $\Longrightarrow c_i < d_i$
  - **Proof** : *Similar to proof of* **LEMMA6.1**
- **LEMMA6.3** : The controller is already monitoring when the train starts to cross.
  - **Lemma** : $\lambda\ s : train(s) = cross \supset$
    $since(\lambda\ s_0 : controller(s_0) = activate)(s) >$
    $since(\lambda\ s_1 : train(s_1) = toCross)(s)$
    $\Longrightarrow (x - b_i) > (x - d_i),\ \forall x : d_i < x < e_i$
    $\Longrightarrow b_i < d_i$
  - **Proof** :
    1. **TC_AX4.1.2** : $d_i - a_i > 2$
    2. From (1) : $d_i - 2 > a_i$
    3. **TC_AX4.3.2** : $b_i - a_i < 1$
    4. From (3) : $b_i - 1 < a_i$
    5. From (2) and (4) : $b_i - 1 < a_i < d_i - 2$

6. From (5) : $b_i < d_i - 1$

7. From (6) : $b_i < d_i$

- **LEMMA6.4** : The controller remains monitoring while the train is crossing.
  - **Lemma** : $\lambda\ s : train(s) = leave \supset$
    $since(\lambda\ s_0 : controller(s_0) = activate)(s) >$
    $since(\lambda\ s_1 : train(s_1) = toCross)(s)$
    $\Longrightarrow (x - b_i) > (x - d_i),\ \forall x : e_i < x < f_i$
    $\Longrightarrow b_i < d_i$
  - **Proof** : *Similar to proof of* **LEMMA6.3**
- **LEMMA6.5** : The controller is not deactivated before the train leaves the crossing.
  - **Lemma** : $\lambda\ s : controller(s) = deactivate \supset$
    $since(\lambda\ s_0 : train(s_0) = cross)(s) >$
    $since(\lambda\ s_1 : controller(s_1) = monitor)(s)$
    $\Longrightarrow (x - e_i) > (x - f_i),\ \forall x, f_i < x < g_i$
    $\Longrightarrow e_i < f_i$
  - **Proof** : (*Invariance Axiom*)
    1. **ACT3.3, ACT3.4** : Exit message is sent after internal event Out.
    2. **SYN_AX5.2** : train sends Exit message to controller.
- **LEMMA6.6** : The gate is not raised before the controller is deactivated.
  - **Lemma** : $\lambda\ s : gate(s) = toOpen \supset$
    $since(\lambda\ s_0 : controller(s_0) = monitor)(s) >$
    $since(\lambda\ s_1 : gate(s_1) = closed)(s)$
    $\Longrightarrow (x - f_i) > (x - g_i),\ \forall x : g_i < x < h_i$
    $\Longrightarrow f_i < g_i$
  - **Proof** :
    1. **TC_AX4.4** : $0 < g_i - f_i < 1$
    2. From (1) : $f_i < g_i$
- **LEMMA6.7** : The gate is not raised before the train leaves the crossing.
  - **Lemma** : $\lambda\ s : gate(s) = toOpen \supset$
    $since(\lambda\ s_0 : train(s_0) = cross)(s) >$
    $since(\lambda\ s_1 : gate(s_1) = closed)(s)$
    $\Longrightarrow (x - e_i) > (x - g_i),\ \forall x : g_i < x < h_i$
    $\Longrightarrow e_i < g_i$

* **Proof :**

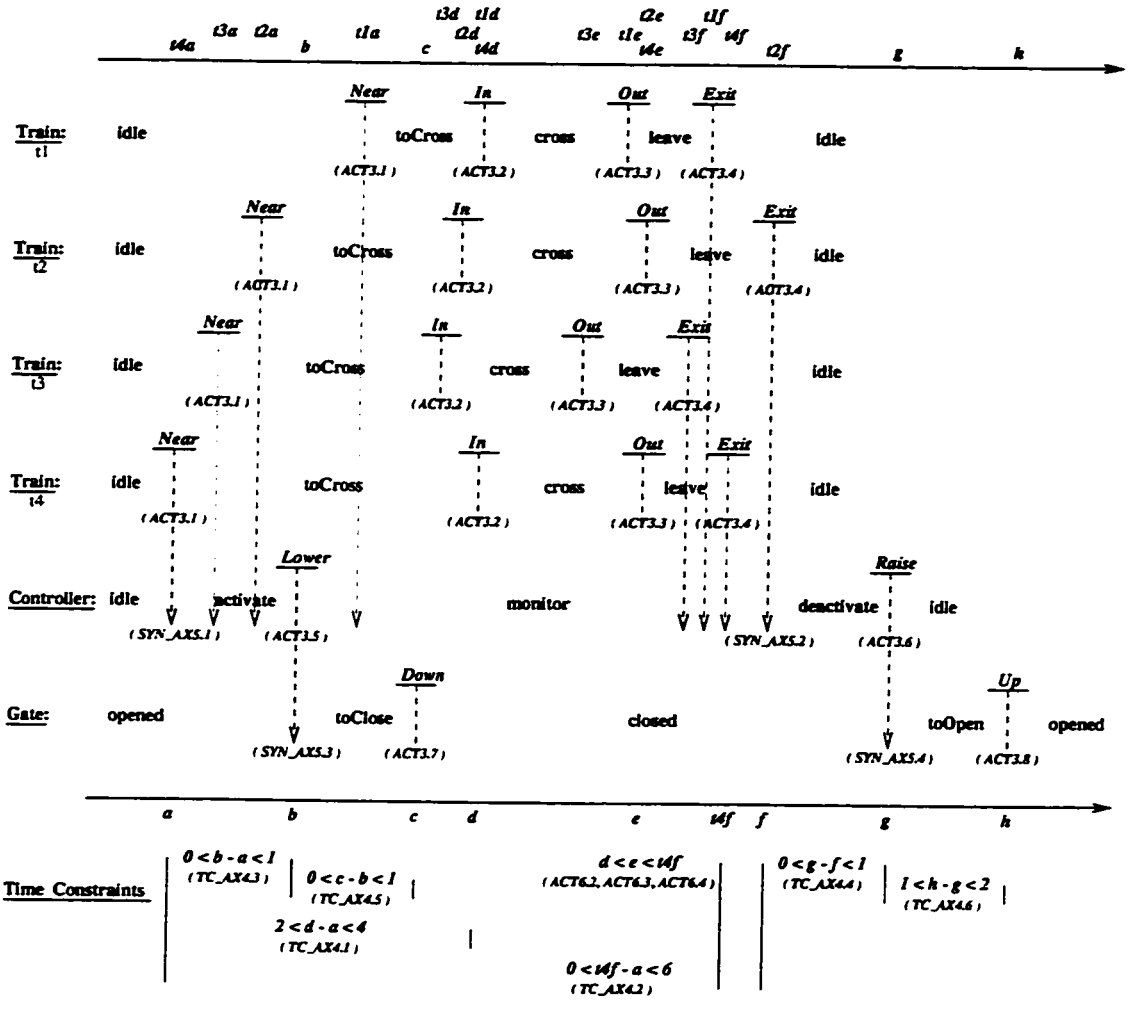
1. $\lambda\ s : gate(s) = toOpen \supset since(\lambda\ s_0 : train(s_0) = cross)(s) >$
   $since(\lambda\ s_1 : controller(s_1) = monitor)(s)$
   $\Longrightarrow (x - e_i) > (x - f_i),\ \forall x, g_i < x < h_i$
   $\Longrightarrow e_i < f_i$
2. Proof of (1) is similar to proof of **LEMMA6.5**
3. From (1) : $e_i < f_i$
4. **LEMMA6.6** : $f_i < g_i$
5. From (3) and (4) : $e_i < f_i < g_i$
6. From (5) : $e_i < g_i$

This completes the verification of safety property for the simple case of one train, one controller, and one gate.

6.5 Verifying a Generalized Railroad Crossing System

In this section, we describe how the verification methodology can be applied to a generalized version of the Train-Gate-Controller system. We use the time-chart in Figure 21 to illustrate the possible scenarios in the system. Considering the system introduced in Figure 8, it is observed that each controller is independent of the other controllers. Similarly, since a gate is controlled by a single controller, we can assert that each gate is independent of the other gates. We can therefore verify the safety property for one controller-gate pair, while considering all the trains in the system. Since all objects of the *Gate* class have the same characteristics, and similarly, all objects of the *Controller* class have the same characteristics, we can conclude that if an arbitrary controller-gate pair operates safely, then all the other controller-gate pairs are safe.

As shown in Figure 21, when the controller is in the idle state, and it receives the Near message from a train (illustrated by train t_4 in the figure), it enters the activate state. When the controller receives a subsequent Near message from any other train, it will be either in the activate state (trains t_2 and t_3 in the figure), or in the monitor state (train t_1 in the figure), and it will remain in the respective state. The count of the number of trains in the crossing is obtained from the attribute inSet (defined in the *Controller* class), which represents an instance of the *Set* trait holding the port id's for messages received from trains. The controller remains idle as long as inSet is empty. It the *Controller* is in the monitor state, the controller is deactivated only when inSet becomes empty. In the inequalities that follow, the subscript i indicates that the gate is being closed for the *i-th* time.



Legend : a, b, c, d, e, f, g, h, t1a, t2a, t3a, t4a, t1d, t2d, t3d, t4d, t1e, t2e, t3e, t4e, t1f, t2f, t3f, t4f : absolute times

Figure 21: Time chart for generalized version of Train-Gate-Controller system.

The axioms characterize the behavior of a general Train-Gate-Controller system. That is, the axioms remain independent of the number of trains, controllers, and gates. The only constraint is that there is a one-to-one correspondence between a controller and the gate controlled by it. Assuming that there are n trains in the system, we denote them with the labels $t_1, t_2, t_3, \ldots, t_n$. Then, without loss of generality, we may assume that the controller receives the **Near** message from train t_k (illustrated by train t_4 in the figure) while it is in the **idle** state, where t_k is the first train to approach the crossing. It receives the **Near** message from trains t_j when it is either in the **activate** state or in the **monitor** state, where

$$1 \leq j \leq n \quad \wedge \quad j \neq k$$

Similarly, we may assume that the controller receives the Exit message from train $t_{k'}$ (illustrated by train t_2 in the figure) causing the attribute inSet to become an empty set, and a transition to the deactivate state, where $t_{k'}$ is the last train to leave the crossing. The controller receives the Exit message from trains $t_{j'}$ when it is in the monitor state, where

$$1 \leq j' \leq n \quad \wedge \quad j' \neq k'$$

Table 3 gives the absolute times at which an event occurs in the generalized version of the Train-Gate-Controller system.

Event	Absolute Time	Description
Near	$t_j a_i$	Message from train j to controller
Near	a_i	Controller goes from idle to activate
Lower	b_i	Message from controller to gate
Down	c_i	Internal to gate
In	$t_j d_i$	Internal to train j
Out	$t_j e_i$	Internal to train j
Exit	$t_j f_i$	Message from train j to controller
Exit	f_i	Controller goes from monitor to deactivate
Raise	g_i	Message from controller to gate
Up	h_i	Internal to gate

Table 3: Absolute times at which the events occur in generalized system.

Therefore,

$$a_i = t_k a_i \quad \wedge \quad t_j a_i \geq a_i, \quad \forall j \in \{x \mid 1 \leq x \leq n\}$$

where a_i represents the value of a at the *i-th* gate closing, and $t_j a_i$ represents the time at which train j starts approaching the gate to cause the event of the *i-th* gate closing. From axioms 4.3.2 and 4.5.2, we have

$$c_i < a_i + 2$$

From axiom 4.1.2, we have

$$d_i - a_i > 2 \quad \wedge \quad t_j d_i - t_j a_i > 2, \quad \forall j \in \{x \mid 1 \leq x \leq n\}$$

From these inequalities, we can prove that

$$t_j d_i > c_i, \quad \forall j \in \{x \mid 1 \leq x \leq n\}$$

We have thus shown that all the trains enter the crossing only after the gate is closed. Similarly, we show that the gate remains closed as long as there is at least one train in

the crossing. As shown in Figure 21, the controller goes in the **deactivate** state only when the last train leaves the crossing (illustrated by train t_2 in the figure). When the controller receives an **Exit** message from any other train, it remains in the **monitor** state (trains t_1, t_3 and t_4 in the figure). Note that the last train to leave the crossing can be any one of t_1, t_2, t_3, and t_4. Therefore,

$$f_i = t_k f_i \quad \wedge \quad t_j f_i \leq f_i, \quad \forall j \in \{x \mid 1 \leq x \leq n\}$$

From axiom 4.4, we have

$$f_i < g_i$$

From these inequalities, we can prove that

$$t_j f_i < g_i, \quad \forall j \in \{x \mid 1 \leq x \leq n\}$$

We have thus shown that the gate enters the state **toOpen** only after all the trains have left the crossing. The cases illustrated by trains t_1, t_2, t_3 and t_4 in the figure capture all the possible scenarios both for trains entering and for trains leaving the crossing. This completes the verification of safety property for the generalized version of the Train-Gate-Controller system.

Chapter 7

Conclusions and Future Work

We have introduced two components of TROMLAB *software development environment supporting a process model for real-time reactive systems. We described the design and functionalities of an animation tool to allow modeling, design, specification and validation, and presented a methodology for formal verification of safety and liveness properties. Future goals include extending the animation tool to support inheritance, a graphical user interface, a methodology for mechanized verification, and extending the* TROM *formalism to encompass multimedia and hybrid systems.*

7.1 Conclusions

The thesis has made two important contributions for the development of real-time reactive systems:

- simulation within TROMLAB development environment, supporting the enhanced iterative process model; and
- verification methodology for TROM-based systems.

The process model supports formalization of requirements while the features, controls, and environmental properties may keep changing. The primary advantage of this model is its support for incremental system development without sacrificing rigor.

The TROMLAB development environment supports the development of real-time reactive systems based on TROM formalism [Ach95], and incorporates object-oriented design principles and formal verification related to PVS [ORS92]. The real benefit of formalizing a design is that it makes automatic analysis possible. TROMLAB environment incorporates a specification language, a type-checker, an interpreter, a simulator, an axiom generator, validation and verification tools, and a visual interface.

The grammar of the specification language is based on the TROM formalism. While being concise, it supports the description of timing constraints, transition specifications with logical assertions, abstract data structures, system configurations allowing object collaboration and synchronization. The interpreter performs syntactic and semantic analysis of TROM class and subsystem configuration specifications, allowing strong type-checking. The simulator supports the validation of TROM subsystems; it includes facilities for debugging the specifications, and for analyzing the trace of the simulated scenario. The axiom generator instantiates the axioms for TROM objects, allowing formal analysis of the behavior of the system under development. We have designed and implemented the interpreter, the simulator, and the axiom generator.

We have also introduced a methodology for the formal verification of TROM subsystems. The methodology has been illustrated for verifying safety properties of a generalized railroad crossing problem. We aim at mechanizing the verification process, using the PVS reasoning system as a back-end.

The contributions of this work include the execution of formal specifications without implementation. The simulator also supports execution of LSL traits to observe their behavior. When used in industrial contexts, this approach reduces maintenance and revision costs due to design errors, thereby promoting the dependability of the system and an overall reduction of development costs in the life-cycle of the system.

TROMLAB environment allows:

- various levels of abstraction for the reactive entity, the reactive system, and the data structures;
- design-time debugging and system validation; and
- formal verification to guarantee safety.

7.2 Future Work

Some of the important directions for future work are the following:

- A graphical user interface is necessary to visually project the simulation process. In addition the user interface will include a browser linking the vast amount of information, such as TROM objects, subsystems, initial requirements, revisions made on the designs, specifications at LSL layer, simulated scenarios, and verification steps. The work in this direction is in progress. The user interface, when completed, will connect all the components and stages of TROMLAB environment.
- The verification methodology should be mechanized and there is good justification for doing this. It is also quite challenging to mechanize the verification methodology described in this thesis. The usefulness of PVS as a back-end engine should be investigated. Some preliminary work in this direction has already been completed. This work will be taken up by me as part of my doctoral work.
- TROM formalism may have to be extended and enriched to meet the modeling requirements of hybrid and multimedia systems. TROMLAB capabilities need to be extended to accommodate systems built on this extended TROM formalism. Once again, I will take this up as part of my doctoral work.
- There is no object-oriented language to handle real-time features in the design. I will explore a two-pronged approach here - to study language design issues for implementing TROM based systems, and to study means of decoupling TROM functionalities and their real-time constraints into different layers of implementation.
- A static analyzer can be used for detecting deadlock and some mechanized assistance included for verifying safety properties. We are working towards achieving these in the following manner:

- An axiom generator will automatically translate the internal representation of a TROM specification into a set of axioms describing the behavior of the TROM. The axioms are to be encoded in the specification language of PVS [ORS92]. These axioms, based on events, states, attributes, transitions, timing constraints, synchronization, and port links, follow from the axiomatization given in [Ach95]. The computational steps of TROMs are specified as predicates on states of the system, while timing constraints are specified as axioms. Translating TROM specifications into PVS specifications makes use of the computational model and the since operator introduced by Shankar [Sha93b].
- Safety properties for the system being developed are specified as predicates on the state of the system. Invariance assertions, lemmas, and theories are used to verify the safety properties using the PVS verification system.

Bibliography

[AAM96] V. S. Alagar, R. Achuthan, and D. Muthiayen. TROMLAB: A software development environment for real-time reactive systems. Technical report, Department of Computer Science, Concordia University, Montréal, Canada, October 1996. Submitted for publication in *ACM Transactions on Software Engineering and Methodology.*

[AAR94a] R. Achuthan, V. S. Alagar, and T. Radhakrishnan. A formal methodology for object-oriented development of real-time reactive systems. In *Workshop on Object-Oriented Real-Time Systems, OOPSLA'94*, Portland, Oregon, October 1994.

[AAR94b] R. Achuthan, V. S. Alagar, and T. Radhakrishnan. An object-oriented framework for specifying reactive systems. In *Workshop on Object-Orientation in Databases and Software Engineering, ACFAS*, Montréal, Canada, May 1994.

[AAR95a] R. Achuthan, V. S. Alagar, and T. Radhakrishnan. A formal model for specification and verification of real-time reactive systems. In *Workshop on Models and Proofs for Concurrent and Real-Time Systems*, Bordeaux, France, June 1995.

[AAR95b] R. Achuthan, V. S. Alagar, and T. Radhakrishnan. An object-oriented modeling of real-time robotic assembly system. In *Proceedings of IEEE First International Conference on Engineering of Complex Computer Systems, ICECCS'95*, Florida, October 1995.

[AAR95c] R. Achuthan, V. S. Alagar, and T. Radhakrishnan. TROM - an object model for reactive system development. In *The 1995 Asian Computing Science Conference, ASIAN'95*, Thailand, December 1995.

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

[Agh86] G. A. Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, Cambridge, Mass., 1986.

[All84] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23, 1984.

[AM96a] V. S. Alagar and D. Muthiayen. OBJ3 User Manual. Technical report, Department of Computer Science, Concordia University, Montréal, Canada, March 1996.

[AM96b] V. S. Alagar and D. Muthiayen. VDM-SL specifications for a simulation tool. Technical report, Department of Computer Science, Concordia University, Montréal, Canada, February 1996. Available from IFAD VDM Examples Repository, URL: http://www.ifad.dk/examples/examples.html.

[AMA96a] V. S. Alagar, D. Muthiayen, and R. Achuthan. Animating real-time reactive systems. In *Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems. ICECCS'96*, Montréal, Canada, October 1996.

[AMA96b] V. S. Alagar, D. Muthiayen, and R. Achuthan. An early exposé to TROMLAB environment. Technical report, Department of Computer Science, Concordia University, Montréal, Canada, July 1996. Submitted for publication in Proceedings of The 1996 Asian Computing Science Conference, ASIAN'96, Singapore, December 1996.

[AMP96] V. S. Alagar, D. Muthiayen, and K. Periyasamy. VDM-SL specification of a graph editor. Technical report, Department of Computer Science, Concordia University, Montréal, Canada, February 1996. Available from IFAD VDM Examples Repository, URL: http://www.ifad.dk/examples/examples.html.

[AR91] V. S. Alagar and G. Ramanathan. Functional specification and proof of correctness for time dependent behaviour of reactive systems. *Formal Aspects of Computing*, 3:253–283, 1991.

[BCC+95] W. B. Butler, J. L. Caldwell, V. A. Carreno, C. M. Holloway, P. S. Miner, and B. L. Di Vito. NASA Langley's research and technology transfer program in formal methods. In *Tenth Annual Conference on Computer Assurance, COMPASS'95*, Gaithersburg, MD, June 1995.

[Bes91] A. Bestaros. Specification and verification of real-time embedded systems using the time-constrained reactive automata. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, pages 240–243, San Antonio, Texas, December 1991.

[BG92] T. E. Bihari and P. Gopinath. Object-oriented real-time systems: Concepts and examples. *IEEE Computer*, 25(12):25–32, December 1992.

[BH95a] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[BH95b] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.

[Boo91] G. Booch. *Object-oriented design with applications.* Benjamin Cummings Pub. Co., Redwood City, California, 1991.

[BRS93] S. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *ACM SIGPLAN-SIGACT Symposium on Programming Languages*, pages 85–98, 1993.

[BS93a] J. P. Bowen and V. Stavridou. The industrial take-up of formal methods in safety-critical and other areas: A perspective. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 183–195, Odense, Denmark, April 1993. Springer-Verlag.

[BS93b] J. P. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.

[But93] R. W. Butler. *An Elementary Tutorial on Formal Specification and Verification Using PVS.* Technical report, National Aeronautics and Space Administration, Langley Research Center, Hampton, VA 23681, September 1993. NASA Technical Memorandum 108991.

[CB91] Johnson S. C. and R. W. Butler. Design for validation. Technical report, NASA Langley Research Center, Hampton, VA, 1991. Presented at the 10th Digital Avionics Systems Conference (DASC), Los Angeles, Ca, Oct 7–11, 1991.

[CDV96] J. Crow and B. L. Di Vito. Formalizing space shuttle software requirements. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice, FMSP'96*, pages 40–48, San Diego. California, January 1996.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[CM92] J. A. Carruth and J. Misra. Proof of a real-time mutual-exclusion algorithm. *Notes on UNITY*, pages 32–92, 1992.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. *A Tutorial Introduction to PVS.* Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, April 1995. Presented at WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida.

[Dol95] A. Dold. Representing, verifying and applying software development steps using the PVS system. In *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95*, Montréal, Canada, July 1995.

[DV96] B. L. Di Vito. Formalizing new navigation requirements for NASA's space shuttle. In *Formal Methods Europe, FME'96*, pages 160–178, Oxford, England, March 1996.

[EC95] W. M. Elseaidy and R. Cleaveland. A tool for modeling and verifying real-time systems. In *Proceedings of IEEE First International Conference on Engineering of Complex Computer Systems, ICECCS'95*, Florida, October 1995.

[FS93] Y. A. Feldman and H. Schneider. Simulating reactive systems by deduction. *ACM Transactions on Software Engineering and Methodology*, 2(2):128–175, April 1993.

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

[GM93] D. Gangopadhyay and S. Mitra. Objchart: Tangible specification of reactive object behavior. In *European Conference on Object Oriented Programming - 93*, 1993.

[Gro95a] The VDM-SL Tool Group. The IFAD VDM-SL Language. Technical Report IFAD-VDM-1, IFAD, The Institute of Applied Computer Science, Denmark, June 1995.

[Gro95b] The VDM-SL Tool Group. The VDM C++ Library. Technical Report IFAD-VDM-7, IFAD, The Institute of Applied Computer Science, Denmark, October 1995.

[Gro95c] The VDM-SL Tool Group. User Manual for the IFAD VDM-SL Toolbox. Technical Report IFAD-VDM-4, IFAD, The Institute of Applied Computer Science, Denmark, October 1995.

[Gro95d] The VDM-SL Tool Group. User Manual for the *VDM-SL$_{to}$C++* Code Generator. Technical Report IFAD-VDM-6, IFAD, The Institute of Applied Computer Science, Denmark, October 1995.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HB96] C. M. Holloway and R. W. Butler. Impediments to industrial use of formal methods. *IEEE Computer*, pages 25–26, April 1996.

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

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hoo94] J. Hooman. Correctness of real time systems by construction. In *Proceedings of the Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 19–40. Springer Verlag, 1994.

[Hoo95] J. Hooman. Using PVS for an assertional verification of the RPC-memory specification problem. Technical report, Department of Mathematics and Computing Science. Eindhoven University of Technology, Eindhoven, The Netherlands, November 1995.

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

[JM94] F. Jahanian and A. Mok. Modecharrt: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[Jon90] C. B. Jones. *Systematic software development using VDM.* Prentice Hall, New York, 1990.

[KSJ89] R. Kurki-Suonio and H. Jarvinen. Action system approach to the specification and design of distributed systems. In *Proceedings of Fifth International Workshop on Software Specification and Design. ACM Software Engineering Notes*, volume 14, pages 34–40, May 1989.

[Lam91] L. Lamport. Temporal logic of actions. Technical Report 79, Systems Research Center, December 1991.

[LT87] N. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th Annual ACM Symposium on Principles of Distributed Computing*, 1987.

[Mey88] B. Meyer. *Object-oriented software construction.* Prentice-Hall, Englewood Cliffs, N.J., 1988.

[MMP91] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editor, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 447–484. Springer Verlag, 1991.

[MSP94] A. Morzenti and P. San Pietro. Object-oriented logical specification of time-critical systems. *ACM Transactions on Software Engineering and Methodology*, 3(1):56–98, January 1994.

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

[ORSvH95] S. Owre, J. M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[OS95] S. Owre and N. Shankar. The formal semantics of PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, October 1995.

[OSR93a] S. Owre, N. Shankar, and J. M. Rushby. The PVS specification language (beta release). Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, April 1993.

[OSR93b] S. Owre, N. Shankar, and J. M. Rushby. User guide for the PVS specification and verification system (beta release). Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, March 1993.

[Ost89] J. S. Ostroff. *Temporal logic for real-time systems.* Research Studies Press Limited, England, 1989.

[Ost94] J. S. Ostroff. Specifying and verifying real-time reactive systems in TTM/RTTL. Technical Report CS-ETR-94-08, Department of Computer Science, York University, Ontario, Canada, September 1994.

[Ost95] J. S. Ostroff. Abstraction and composition of discrete real-time systems. Technical Report CS-ETR-95-02, Department of Computer Science, York University, Ontario, Canada, October 1995.

[Par95] D. L. Parnas. Fighting complexity. In *Proceedings of IEEE First International Conference on Engineering of Complex Computer Systems, ICECCS'95*, Florida, October 1995.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, New Jersey, 1991.

[RDdM+95] A. Rendon, J. C. Duenas, M. A. de Miguel, J. Leskela, J. A. de la Puente, G. Leon, and A. Alonso. Animation of heterogeneous prototypes of real-time systems. In *Proceedings of IEEE First International Conference on Engineering of Complex Computer Systems, ICECCS'95*, pages 47–54, Florida, October 1995.

[Rus92] J. Rushby. Formal methods for dependable real-time systems. In *International Symposium on Real-Time Embedded Processing for Space Applications*, pages 355–366, Les Saintes-Maries-de-la-Mer, France, November 1992. CNES, the French Space Agency, Cépaduès-Éditions, Toulouse, France.

[Rus93] J. Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, California, December 1993.

[Rus94] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[Rus95] J. Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, California, March 1995.

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

[Sha93a] N. Shankar. Abstract datatypes in PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, December 1993.

[Sha93b] N. Shankar. Mechanized verification of real-time systems using PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, March 1993.

[Sha93c] N. Shankar. Verification of real-time systems using PVS. In Costas Courcoubetis, editor, *Proceedings of Computer Aided Verification, CAV'93*, volume 697 of *Lecture Notes in Computer Science*, pages 280–291, Elounda, Greece, June 1993. Springer Verlag.

[Ska94] J. U. Skakkebaek. *A verification assistant for a real-time logic*. PhD thesis, Department of Computer Science, Technical University of Denmark, Denmark, November 1994.

[SKS90a] K. Systa and R. Kurki-Suonio. The DisCo language and temporal logic of actions. Technical report, Software Systems Laboratory, Tampere University of Technology, Tampere, Finland, 1990.

[SKS90b] K. Systa and R. Kurki-Suonio. DisCo specification language: marriage of actions and objects. Technical report, Software Systems Laboratory, Tampere University of Technology, Tampere, Finland, 1990.

[SKS92] K. Systa and R. Kurki-Suonio. Modeling of distributed real-time systems in DisCo. In *Euromicro'92 Workshop on Real-Time Systems*, Athens, Greece, June 1992.

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. The PVS proof checker: A reference manual (beta release). Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, March 1993.

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

[TMM88] M. Tuttle, M. Meritt, and F. Modugno. Time constrained automata. Technical report, MIT/LCS, November 1988.

[VH96] J. Vitt and J. Hooman. Assertional specification and verification using PVS of the steam boiler control system. Technical report, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, January 1996.

[ZAK95] J. Ziegler, M. Awad, and J. Kuusela. Applying object-oriented technology in real-time systems with the OCTOPUS method. In *Proceedings of IEEE First International Conference on Engineering of Complex Computer Systems*, pages 306–309, October 1995.

Appendix A

Class Diagrams in OMT Notation

The OMT (Object Modeling Technique) notation allows describing object-oriented designs using class diagrams and state diagrams. The object model, the functional model, and the dynamic model of a system can be represented in this notation. We use detailed class diagrams to illustrate the design of the TROM, *Subsystem, and Simulation Event classes.*

TROM

CLASS DIAGRAM

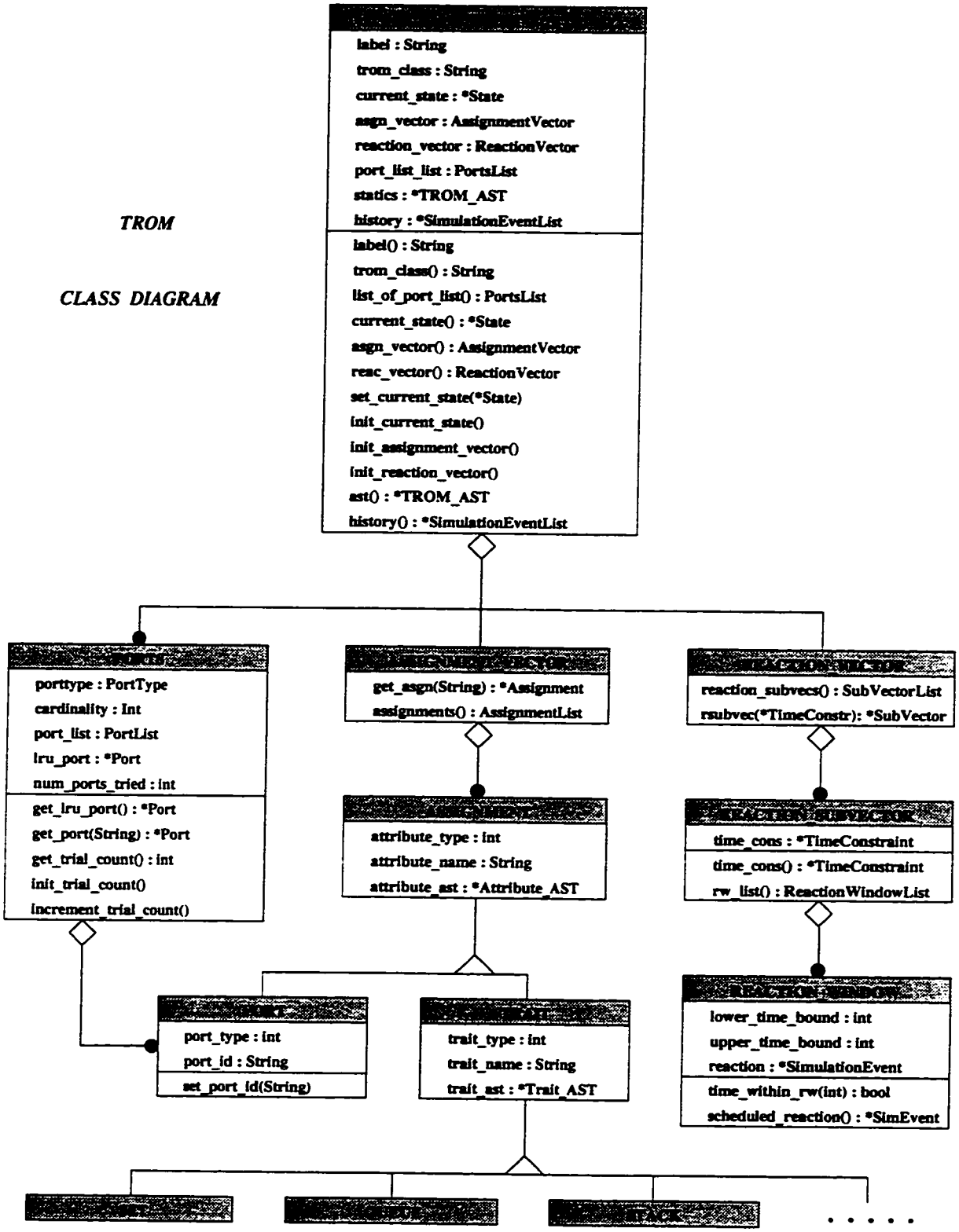


Figure 22: Detailed class diagram illustrating TROM object model.

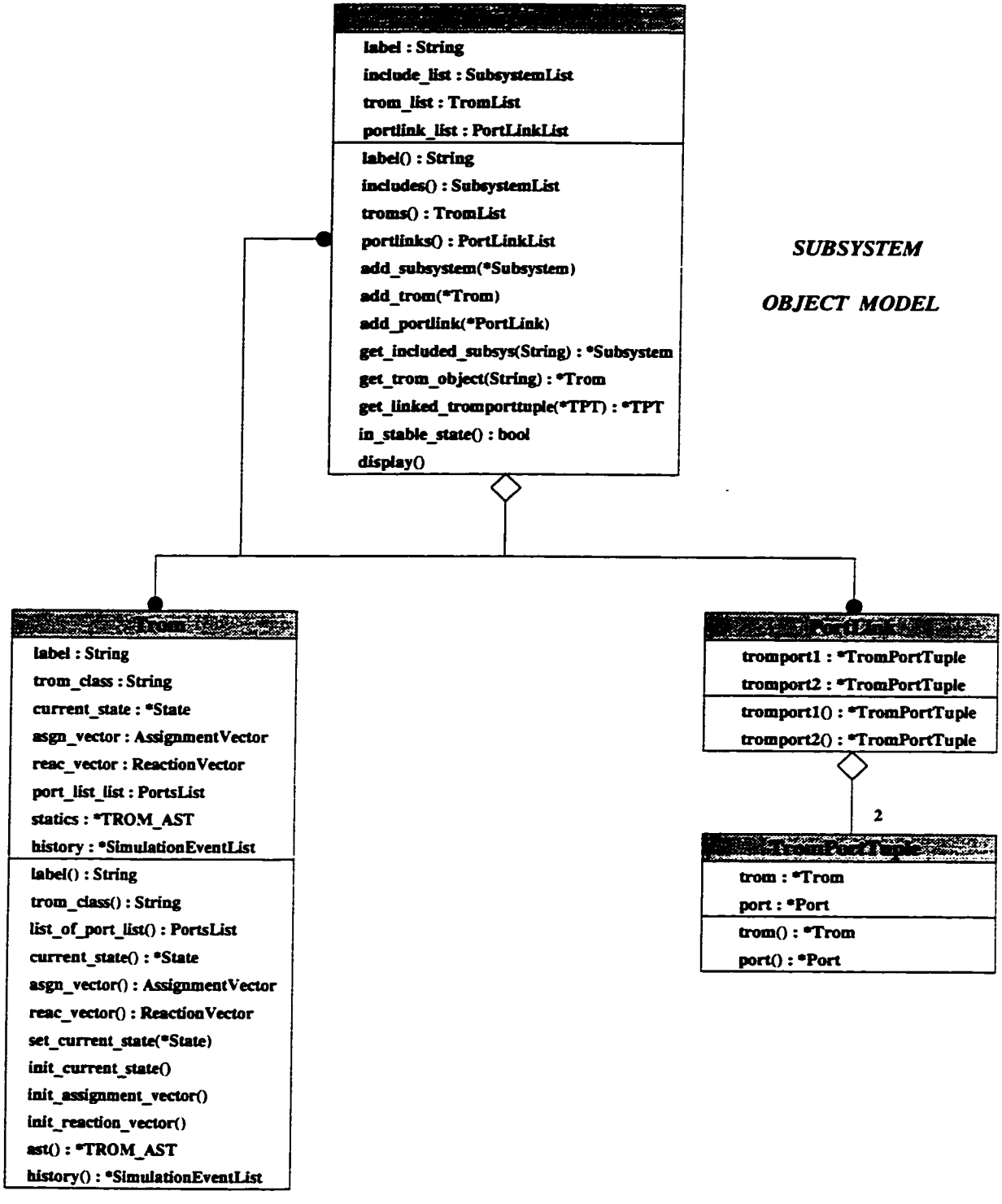


Figure 23: Detailed class diagram illustrating Subsystem object model.

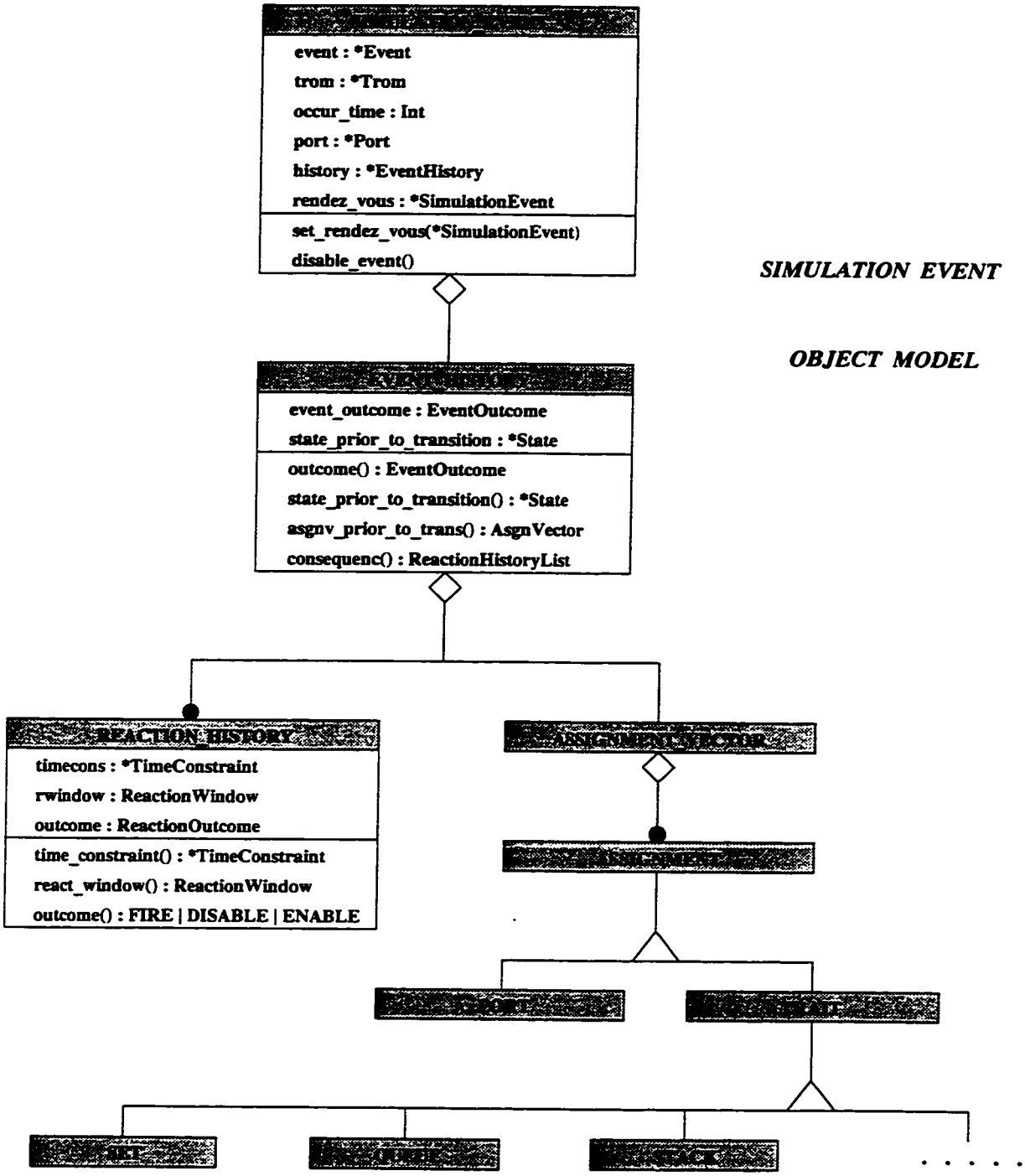


Figure 24: Detailed class diagram illustrating Simulation Event object model.

Appendix B

VDM Specifications

The specification language of VDM (Vienna Development Model) incorporates several constructs allowing formal specification, and refinement. We have provided two sets of specifications for the functionalities of the simulator in VDM-SL. In the first version (Section B.1), we use implicit operations to describe the steps in the simulation. We then rewrite most of the operations using explicit operations (Section B.2). Pre- and post-conditions are included to specify the effect of the operations on the state of the system. This exercise has allowed us to ensure that the simulator conforms to the operational semantics of TROM. *We have derived an implementation in C++ from the formal specifications.*

B.1 Specification of Simulator Using Implicit Operations

module *TROM*

exports all

definitions

types

```
1.0  PortType :: label : String
 .1              cardinality : N
 .2              portlist : Port*

 .3  inv mk-PortType (label, cardinality, portlist) ≜
 .4      (cardinality = card elems portlist);

2.0  Event :: label : String
 .1           type : EventType
 .2           porttype : PortType

 .3  inv mk-Event (label, type, porttype) ≜
 .4      (((type = INTERNAL) ∧ (porttype.label = "NULLPORT")) ∨
 .5       ((type = INPUT) ∧ (porttype.label ≠ "NULLPORT")) ∨
 .6       ((type = OUTPUT) ∧ (porttype.label ≠ "NULLPORT")));

3.0  State :: label : String
 .1           type : StateType
 .2           isinitial : B
 .3           substates : State-set

 .4  inv mk-State (label, type, isinitial, substates) ≜
 .5      (let exists-entry-state : State-set → B
 .6           exists-entry-state (substates) ≜
 .7             ((∃! s ∈ substates · (s.isinitial = true)) ∧
 .8              (∀ s ∈ substates ·
 .9                   (((s.type = SIMPLE) ∧ (s.substates = {})) ∨
.10                    ((s.type = COMPLEX) ∧ (exists-entry-state (s.substates)))))) in
.11      (((type = SIMPLE) ∧ (substates = {})) ∨
.12       ((type = COMPLEX) ∧ (exists-entry-state (substates)))));

4.0  Attribute :: label : String
 .1               type : String;
```

5.0 *LSLTrait* :: *traitlabel* : *String*

.1 *traittype* : *String*

.2 *elementtypes* : *String**;

6.0 *AttrFunction* :: *stat* : *State*

.1 *attributes* : *Attribute*-set;

7.0 *TransitionSpec* :: *label* : *String*

.1 *sourcestate* : *State*

.2 *destinstate* : *State*

.3 *triggerevent* : *Event*

.4 *portcondition* : $\mathbb{B}$

.5 *enabcondition* : $\mathbb{B}$

.6 *postcondition* : $\mathbb{B}$;

8.0 *TimeConstraint* :: *label* : *String*

.1 *transition* : *TransitionSpec*

.2 *constrainedevent* : *Event*

.3 *timebounds* : *ReactionWindow*

.4 *disablingstates* : *State*-set

.5 *reactionwindows* : *ReactionWindow*-set

.6 inv mk-*TimeConstraint* (*label*, *transition*, *cevent*, *tbounds*, *dstates*, *rwindows*) $\triangleq$

.7 $(((cevent.type = \text{INTERNAL}) \vee (cevent.type = \text{OUTPUT})) \wedge$

.8 $(\forall rw \in rwindows \cdot$

.9 $((rw.uppertimebound - rw.lowertimebound) =$

.10 $(tbounds.uppertimebound - tbounds.lowertimebound))))$;

9.0 *EventType* = INPUT | INTERNAL | OUTPUT;

10.0 *StateType* = SIMPLE | COMPLEX;

11.0 *ReactionWindow* :: *lowertimebound* : $\mathbf{N}$

.1 *uppertimebound* : $\mathbf{N}$

.2 inv mk-*ReactionWindow* (*lowertimebound*, *uppertimebound*) $\triangleq$

.3 $(lowertimebound \leq uppertimebound)$;

12.0 *Port* :: *label* : *String*;

13.0 $PortLink :: tromporttuple1 : TromPortTuple$

.1 $\quad tromporttuple2 : TromPortTuple$

.2 inv mk-$PortLink$ ($tromporttuple1, tromporttuple2$) $\triangleq$

.3 $\quad (tromporttuple1.tromlabel \neq tromporttuple2.tromlabel)$;

14.0 $TromPortTuple :: tromlabel : String$

.1 $\quad portlabel : String$;

15.0 $SimulationEvent :: eventlabel : String$

.1 $\quad tromlabel : String$

.2 $\quad portlabel : String$

.3 $\quad occurtime : \mathbb{N}$

.4 $\quad eventhistory : [EventHistory]$;

16.0 $EventHistory :: triggeredtransition : \mathbb{B}$

.1 $\quad tromcurrentstate : [State]$

.2 $\quad assignmentvector : [\text{token}]$

.3 $\quad reactionshistory : ReactionHistory\text{-set}$;

17.0 $ReactionHistory :: timeconstraint : TimeConstraint$

.1 $\quad reactionwindow : ReactionWindow$

.2 $\quad reaction : Reaction$;

18.0 $Reaction = \text{FIRED} \mid \text{DISABLED} \mid \text{ENABLED}$;

19.0 $LSLTraitDefinition :: label : String$

.1 $\quad paramets : String^*$;

20.0 $String = \text{char}^*$;

21.0 *Trom* :: *label* : *String*

.1 *tromclass* : *String*

.2 *porttypes* : *PortType*-set

.3 *events* : *Event*-set

.4 *states* : *State*-set

.5 *attributes* : *Attribute*-set

.6 *lsltraits* : *LSLTrait*-set

.7 *attrfunctions* : *AttrFunction*-set

.8 *transitionspecs* : *TransitionSpec*-set

.9 *timeconstraints* : *TimeConstraint*-set

.10 *currentstate* : *State*

.11 *assignmentvector* : token

```
.12 inv mk-Trom (label, tromclass, porttypes, events, states, attributes, lsltraits, attrfunctions,
.13                 transitionspecs, timeconstraints, currentstate, assignmentvector) ≜
.14   (∀ pt1, pt2 ∈ porttypes ·
.15       (pt1.label = pt2.label ⇒ pt1 = pt2)) ∧
.16   (∀ e1, e2 ∈ events ·
.17       (e1.label = e2.label ⇒ e1 = e2)) ∧
.18   (∀ s1, s2 ∈ states ·
.19       (s1.label = s2.label ⇒ s1 = s2)) ∧
.20   (∀ a1, a2 ∈ attributes ·
.21       (a1.label = a2.label ⇒ a1 = a2)) ∧
.22   (∀ tr1, tr2 ∈ lsltraits ·
.23       (tr1.traittype = tr2.traittype ⇒ tr1 = tr2)) ∧
.24   (∀ af1, af2 ∈ attrfunctions ·
.25       (af1.stat = af2.stat ⇒ af1 = af2)) ∧
.26   (∀ ts1, ts2 ∈ transitionspecs ·
.27       (ts1.label = ts2.label ⇒ ts1 = ts2)) ∧
.28   (∀ tc1, tc2 ∈ timeconstraints ·
.29       (tc1.label = tc2.label ⇒ tc1 = tc2)) ∧
.30   (∀ e ∈ events ·
.31       (∃ pt ∈ porttypes · (pt = e.porttype))) ∧
.32   (∃! s ∈ states ·
.33       (s.isinitial = true)) ∧
.34   (∃! s ∈ states ·
.35       ((currentstate = s) ∨ (substate-of (currentstate, s)))) ∧
.36   (∀ a ∈ attributes ·
.37       ((∃! pt ∈ porttypes · (pt.label = a.type)) ∨
.38        (∃! tr ∈ lsltraits · (tr.traittype = a.type)))) ∧
.39   (∀ tr ∈ lsltraits ·
.40       (∀ el ∈ elems tr.elementtypes ·
.41           ((∃! pt ∈ porttypes · (pt.label = el)) ∨
.42            (∃! tr2 ∈ lsltraits · (tr2.traittype = el))))) ∧
.43   (∀ af ∈ attrfunctions ·
.44       ((∃! s ∈ states · ((s = af.stat) ∨ (substate-of (af.stat, s)))) ∧
.45        (∀ afa ∈ af.attributes ·
.46            (∃! a ∈ attributes · (a = afa))))) ∧
.47   (∀ ts ∈ transitionspecs ·
.48       ((∃! s ∈ states · ((s = ts.sourcestate) ∨ (substate-of (ts.sourcestate, s)))) ∧
.49        (∃! d ∈ states · ((d = ts.destinstate) ∨ (substate-of (ts.destinstate, d)))) ∧
.50        (∃! e ∈ events · (e = ts.triggerevent)))) ∧
```

.51 $(\forall\, tc \in timeconstraints \cdot$

.52 $((\exists!\, ts \in transitionspecs \cdot (ts = tc.transition)) \wedge$

.53 $(\exists!\, e \in events \cdot$

.54 $((e = tc.constrainedevent) \wedge$

.55 $((e.type = \text{INTERNAL}) \vee (e.type = \text{OUTPUT})))) \wedge$

.56 $(\forall\, ds \in tc.disablingstates \cdot$

.57 $(\exists!\, s \in states \cdot ((s = ds) \vee (substate\text{-}of\,(ds, s)))))));$

22.0 $Subsystem :: label : String$

.1 $includes : Subsystem\text{-set}$

.2 $troms : Trom\text{-set}$

.3 $portlinks : PortLink\text{-set}$

.4 inv mk-$Subsystem\,(label, includes, troms, portlinks) \triangleq$

.5 $(\forall\, s1, s2 \in includes \cdot$

.6 $(s1.label = s2.label \Rightarrow s1 = s2)) \wedge$

.7 (let $included\text{-}subsystem : String \times Subsystem\text{-set} \to \mathbb{B}$

.8 $included\text{-}subsystem\,(subsystemlabel, subsystems) \triangleq$

.9 $(\exists!\, s \in subsystems \cdot$

.10 $((s.label = subsystemlabel) \vee$

.11 $(included\text{-}subsystem\,(subsystemlabel, s.includes)))$ in

.12 $(\neg\, included\text{-}subsystem\,(label, includes))) \wedge$

.13 $(\forall\, trom1, trom2 \in troms \cdot$

.14 $(trom1.label = trom2.label \Rightarrow trom1 = trom2)) \wedge$

.15 (let $included\text{-}trom : String \times Subsystem\text{-set} \to \mathbb{B}$

.16 $included\text{-}trom\,(tromlabel, subsystems) \triangleq$

.17 $(\exists!\, s \in subsystems \cdot$

.18 $((\exists!\, trom \in s.troms \cdot$

.19 $(trom.label = tromlabel)) \vee$

.20 $(included\text{-}trom\,(tromlabel, s.includes)))$ in

.21 $(\forall\, trom \in troms \cdot$

.22 $(\neg\, included\text{-}trom\,(trom.label, includes)))) \wedge$

```
.23   (let linked-trom : TromPortTuple × Trom-set → 𝔹
.24       linked-trom (tptuple, troms) ≙
.25         (∃! trom ∈ troms ·
.26               ((trom.label = tptuple.tromlabel) ∧
.27                (∃! pt ∈ trom.porttypes ·
.28                      (∃! p ∈ elems pt.portlist · (p.label = tptuple.portlabel))))),
.29       linked-subsystem : TromPortTuple × Subsystem-set → 𝔹
.30       linked-subsystem (tptuple, subsystems) ≙
.31         (∃! s ∈ subsystems ·
.32               (((linked-trom (tptuple, s.troms)) ∧
.33                 (¬ linked-subsystem (tptuple, s.includes))) ∨
.34                ((¬ linked-trom (tptuple, s.troms)) ∧
.35                 (linked-subsystem (tptuple, s.includes)))) ∧
.36               (∀ s2 ∈ {su | su : Subsystem · su ∈ subsystems ∧ su ≠ s} ·
.37                     ((¬ linked-trom (tptuple, s2.troms)) ∧
.38                      (¬ linked-subsystem (tptuple, s2.includes))))) in
.39   (∀ pl ∈ portlinks ·
.40         ((((linked-trom (pl.tromporttuple1, troms)) ∧
.41           (¬ linked-subsystem (pl.tromporttuple1, includes))) ∨
.42          ((¬ linked-trom (pl.tromporttuple1, troms)) ∧
.43           (linked-subsystem (pl.tromporttuple1, includes)))) ∧
.44          (((linked-trom (pl.tromporttuple2, troms)) ∧
.45            (¬ linked-subsystem (pl.tromporttuple2, includes))) ∨
.46           ((¬ linked-trom (pl.tromporttuple2, troms)) ∧
.47            (linked-subsystem (pl.tromporttuple2, includes)))))))
```

```
23.0  state system of
 .1     SUBSYSTEM : Subsystem
 .2     SIMULATIONEVENTLIST : SimulationEvent*
 .3     LSLLIBRARY : LSLTraitDefinition-set
 .4     CLOCK : N
 .5     inv mk-system (subsystem, simulationeventlist, lsllibrary, clock) ≜
 .6        (let contains-trom : Subsystem → B
 .7             contains-trom (subsys) ≜
 .8               ((subsys.troms ≠ {}) ∨
 .9                (∃! s ∈ subsys.includes · (contains-trom (s)))) in
.10         contains-trom (subsystem)) ∧
.11        (let contains-portlink : Subsystem → B
.12             contains-portlink (subsys) ≜
.13               ((subsys.portlinks ≠ {}) ∨
.14                (∃! s ∈ subsys.includes · (contains-portlink (s)))) in
.15         contains-portlink (subsystem)) ∧
.16        (∀ i, j ∈ inds simulationeventlist ·
.17              (((i = j) ∧
.18                (simulationeventlist (i) = simulationeventlist (j))) ∨
.19               ((i < j) ∧
.20                (simulationeventlist (i).occurtime ≤
.21                 simulationeventlist (j).occurtime)) ∨
.22               ((i > j) ∧
.23                (simulationeventlist (i).occurtime ≥
.24                 simulationeventlist (j).occurtime)))) ∧
.25        (∀ se1, se2 ∈ elems simulationeventlist ·
.26              (((se1.occurtime = se2.occurtime) ∧ (se1.tromlabel ≠ se2.tromlabel)) ∨
.27               (se1.occurtime ≠ se2.occurtime))) ∧
```

```
.28        (let accepted-by-trom : SimulationEvent × Subsystem → 𝔹
.29            accepted-by-trom (se, subsys) ≙
.30              (∃! trom ∈ subsys.troms ·
.31                    ((trom.label = se.tromlabel) ∧
.32                     (∃! e ∈ trom.events ·
.33                           ((e.label = se.eventlabel) ∧
.34                            (∃! pt ∈ trom.porttypes ·
.35                                  ((pt = e.porttype) ∧
.36                                   (∃! p ∈ elems pt.portlist ·
.37                                         (p.label = se.portlabel)))))))) in
.38        let accepted-by-subsystem : SimulationEvent × Subsystem → 𝔹
.39            accepted-by-subsystem (se, subsys) ≙
.40              (∃! s ∈ subsys.includes ·
.41                    ((((accepted-by-trom (se, s)) ∧ (¬ accepted-by-subsystem (se, s))) ∨
.42                      ((¬ accepted-by-trom (se, s)) ∧ (accepted-by-subsystem (se, s)))) ∧
.43                     (∀ s2 ∈ {su | su : Subsystem · su ∈ subsys.includes ∧ su ≠ s} ·
.44                           ((¬ accepted-by-trom (se, s2)) ∧
.45                            (¬ accepted-by-subsystem (se, s2)))))) in
.46        (∀ se ∈ elems simulationeventlist ·
.47              (((accepted-by-trom (se, subsystem)) ∧
.48                (¬ accepted-by-subsystem (se, subsystem))) ∨
.49               ((¬ accepted-by-trom (se, subsystem)) ∧
.50                (accepted-by-subsystem (se, subsystem)))))) ∧
.51       (let exists-lsltrait : Subsystem → 𝔹
.52           exists-lsltrait (subsys) ≙
.53             ((∀ trom ∈ subsys.troms ·
.54                   (∀ tr ∈ trom.lsltraits ·
.55                         (∃ traitdef ∈ lsllibrary · (traitdef.label = tr.traitlabel)))) ∧
.56              (∀ s ∈ subsys.includes · (exists-lsltrait (s)))) in
.57       (exists-lsltrait (subsystem)))
.58    init mk-system (subsys, simeventlist, lsllib, clock) ≙ simeventlist ≠ [] ∧ clock = 0
.59 end
```

functions

24.0 *get-trom-object* (*tromlabel* : *String*, *subsystem* : *Subsystem*) *trom* : [*Trom*]
.1 post ((*trom* ∈ *subsystem.troms* ∧ *trom.label* = *tromlabel*) ∨
.2 (∃! *s* ∈ *subsystem.includes* ·
.3 (*trom* = *get-trom-object* (*tromlabel*, *s*))) ∨
.4 (*trom* = nil)) ;

25.0 *get-transition-spec* (*trom* : *Trom*, *se* : *SimulationEvent*) *ts* : [*TransitionSpec*]
.1 pre (*trom.label* = *se.tromlabel*)
.2 post (((*ts* ∈ *trom.transitionspecs*) ∧
.3 ((*trom.currentstate.label* = *ts.sourcestate.label*) ∨
.4 (*substate-of* (*trom.currentstate*, *ts.sourcestate*))) ∧
.5 (*ts.triggerevent.label* = *se.eventlabel*) ∧
.6 (*ts.portcondition* = true) ∧
.7 (*ts.enabcondition* = true)) ∨
.8 (*ts* = nil)) ;

26.0 *substate-of* : *State* × *State* → 𝔹
.1 *substate-of* (*substate*, *complexstate*) ≙
.2 (*substate* ∈ *complexstate.substates* ∨
.3 (∃! *s* ∈ *complexstate.substates* ·
.4 (*s.type* = COMPLEX ∧ *substate-of* (*substate*, *s*))))
.5 pre (*complexstate.type* = COMPLEX) ;

27.0 *get-entry-state* (*complexstate* : *State*) *entry* : *State*
.1 pre (*complexstate.type* = COMPLEX)
.2 post (∃! *s* ∈ *complexstate.substates* ·
.3 ((*s.isinitial* = true) ∧
.4 ((*s.type* = SIMPLE ∧ *entry* = *s*) ∨
.5 (*s.type* = COMPLEX ∧ *entry* = *get-entry-state* (*s*))))) ;

28.0 *get-initial-state* (*trom* : *Trom*) *initial* : *State*
.1 pre (*trom.states* ≠ {})
.2 post (∃! *s* ∈ *trom.states* ·
.3 ((*s.isinitial* = true) ∧
.4 ((*s.type* = SIMPLE ∧ *initial* = *s*) ∨
.5 (*s.type* = COMPLEX ∧ *initial* = *get-entry-state* (*s*))))) ;

29.0 *get-linked-tromport-tuple* $(tupleA : TromPortTuple, subsystem : Subsystem)$

.1 $\qquad\qquad\qquad\qquad tupleB : [TromPortTuple]$

.2 post $((\exists!\, pl \in subsystem.portlinks \cdot$

.3 $\qquad ((pl.tromporttuple1 = tupleA \wedge pl.tromporttuple2 = tupleB) \vee$

.4 $\qquad (pl.tromporttuple2 = tupleA \wedge pl.tromporttuple1 = tupleB))) \vee$

.5 $\quad (\exists!\, s \in subsystem.includes \cdot$

.6 $\qquad (tupleB = get\text{-}linked\text{-}tromport\text{-}tuple\,(tupleA, s))) \vee$

.7 $\quad (tupleB = \text{nil}\ ))$;

30.0 *exists-in-subsystem* $: Trom \times Subsystem \rightarrow \mathbb{B}$

.1 *exists-in-subsystem* $(trom, subsys) \triangleq$

.2 $\quad (trom \in subsys.troms \vee$

.3 $\quad (\exists!\, subsystem \in subsys.includes \cdot$

.4 $\qquad (exists\text{-}in\text{-}subsystem\,(trom, subsystem))))$

.5 pre $((subsys.troms \neq \{\}) \vee (subsys.includes \neq \{\}))$;

31.0 *get-unconstrained-internal-event* $(trom : Trom)\ event : [Event]$

.1 post $((\exists\, ts \in trom.transitionspecs \cdot$

.2 $\qquad ((ts.sourcestate = trom.currentstate) \wedge$

.3 $\qquad (ts.triggerevent.type = \text{INTERNAL}) \wedge$

.4 $\qquad (\neg\, constrained\text{-}event\,(trom, ts.triggerevent)) \wedge$

.5 $\qquad (event = ts.triggerevent))) \vee$

.6 $\quad (event = \text{nil}\ ))$;

32.0 *constrained-event* $: Trom \times Event \rightarrow \mathbb{B}$

.1 *constrained-event* $(trom, event) \triangleq$

.2 $\quad (\exists\, tc \in trom.timeconstraints \cdot$

.3 $\qquad (tc.constrainedevent = event))$

.4 pre $(event \in trom.events)$;

33.0 *get-simevent-index* $(se : SimulationEvent, se\text{-}list : SimulationEvent^*)\ index : \mathbf{N}_1$

.1 pre $(se \in \text{elems}\ se\text{-}list)$

.2 post $(se\text{-}list\,(index) = se)$;

34.0 *get-random-time-within-rw* $(rw : ReactionWindow)\ time : \mathbf{N}$

.1 post $((time \geq rw.lowertimebound) \wedge (time \leq rw.uppertimebound))$;

35.0 *get-lru-port* (*portlist* : *Port**) *port* : *Port*

.1 pre (*portlist* $\neq$ [])

.2 post (*port* $\in$ elems *portlist*)

operations

36.0 *simulator* : () $\xrightarrow{o}$ ()

.1 *simulator*() $\triangleq$

.2 (dcl $i : \mathbf{N}_1 := 1$;

.3 *initialize-simulation-clock*() ;

.4 *schedule-unconstrained-internal-events-from-initial-state*() :

.5 while ($i \leq$ len *SIMULATIONEVENTLIST*)

.6 do (while (*CLOCK* < *SIMULATIONEVENTLIST* (i).*occurtime*)

.7 do (*update-simulation-clock*());

.8 while (($i \leq$ len *SIMULATIONEVENTLIST*) $\wedge$

.9 (*CLOCK* = *SIMULATIONEVENTLIST* (i).*occurtime*))

.10 do (*handle-event*(*SIMULATIONEVENTLIST* (i)) :

.11 $i := i + 1$)))

.12 pre ((*SIMULATIONEVENTLIST* $\neq$ []) $\wedge$

.13 ($\forall$ *se* $\in$ elems *SIMULATIONEVENTLIST* $\cdot$

.14 ((*se*.*occurtime* $\geq$ *CLOCK*) $\wedge$

.15 (*se*.*eventhistory* = nil))) $\wedge$

.16 ($\forall$ *trom* $\in$ {*trom* | *trom* : *Trom* $\cdot$ *exists-in-subsystem* (*trom*, *SUBSYSTEM*)} $\cdot$

.17 ((*trom*.*currentstate* = *get-initial-state* (*trom*)) $\wedge$

.18 ($\forall$ *tc* $\in$ *trom*.*timeconstraints* $\cdot$

.19 (*tc*.*reactionwindows* = {}))))))

.20 post ((*SIMULATIONEVENTLIST* $\neq$ []) $\wedge$

.21 (*SIMULATIONEVENTLIST* (len *SIMULATIONEVENTLIST*).*occurtime* =

.22 *CLOCK*) $\wedge$

.23 ($\forall$ *se* $\in$ elems *SIMULATIONEVENTLIST* $\cdot$

.24 ((*se*.*occurtime* $\leq$ *CLOCK*) $\wedge$

.25 (*se*.*eventhistory* $\neq$ nil))) $\wedge$

.26 ($\forall$ *trom* $\in$ {*trom* | *trom* : *Trom* $\cdot$ *exists-in-subsystem* (*trom*, *SUBSYSTEM*)} $\cdot$

.27 ($\forall$ *tc* $\in$ *trom*.*timeconstraints* $\cdot$

.28 (*tc*.*reactionwindows* = {}))))) ;

```
37.0  handle-event : SimulationEvent ⟶° ()
 .1   handle-event (se) ≜
 .2     (dcl trom : [Trom],
 .3          ts : [TransitionSpec];
 .4      trom := get-trom-object (se.tromlabel, SUBSYSTEM);
 .5      if trom = nil
 .6      then return
 .7      else skip;
 .8      ts := get-transition-spec (trom, se);
 .9      if ts = nil
 .10     then (update-history-notransition(trom, se, ts) )
 .11     else (update-history-assignment-vector(trom, se, ts) ;
 .12           if ts.postcondition = false
 .13           then (update-history-notransition(trom, se, ts) )
 .14           else (update-history-transition(trom, se, ts) ;
 .15                 update-trom-current-state(trom, se, ts) ;
 .16                 handle-transition(trom, se, ts) ;
 .17                 schedule-unconstrained-internal-event(trom, se) )))
 .18  pre (se.occurtime = CLOCK')
 .19    post (CLOCK' = CLOCK) ;
```

```
38.0  handle-transition : Trom × SimulationEvent × TransitionSpec ⟶° ()
 .1   handle-transition (trom, se, ts) ≜
 .2     (for all tc ∈ trom.timeconstraints
 .3      do (if tc.constrainedevent.label = se.eventlabel
 .4          then (for all rw ∈ tc.reactionwindows
 .5                do (if se.occurtime ≥ rw.lowertimebound ∧
 .6                       se.occurtime ≤ rw.uppertimebound
 .7                    then (update-history-fire-reaction(trom, se, tc, rw) ;
 .8                          fire-reaction(trom. se. tc, rw) )
 .9                    else skip))
.10          else skip:
.11          if trom.currentstate ∈ tc.disablingstates
.12          then (for all rw ∈ tc.reactionwindows
.13                do (update-history-disable-reaction(trom, se, tc, rw) ;
.14                    disable-reaction(trom, se, tc, rw) ))
.15          else skip:
.16          if ts.label = tc.transition.label
.17          then (update-history-enable-reaction(trom. se, tc, ts) :
.18                enable-reaction(trom. se, tc, ts) )
.19          else skip))
.20  pre  (se.occurtime = CLOCK')
.21    post (CLOCK' = \overline{CLOCK'}) ;
```

39.0 *update-trom-current-state* (*trom* : *Trom*, *se* : *SimulationEvent*, *ts* : *TransitionSpec*)

.1 pre (*ts.postcondition* = true)

.2 post (((*ts.destinstate.type* = SIMPLE) ∧

.3 (*trom.currentstate* = *ts.destinstate*)) ∨

.4 ((*ts.destinstate.type* = COMPLEX) ∧

.5 (*trom.currentstate* = *get-entry-state* (*ts.destinstate*)))) ;

40.0 *update-history-assignment-vector* (*trom* : *Trom*, *se* : *SimulationEvent*, *ts* : [*TransitionSpec*])

.1 pre (*ts* ≠ nil)

.2 post (*se.eventhistory.assignmentvector* = *trom.assignmentvector*) ;

41.0 *update-history-notransition* (*trom* : *Trom*, *se* : *SimulationEvent*, *ts* : [*TransitionSpec*])

.1 pre ((*ts* = nil) ∨ (*ts.postcondition* = false))

.2 post ((*se.eventhistory.triggeredtransition* = false) ∧

.3 (*se.eventhistory.tromcurrentstate* = nil) ∧

.4 (*se.eventhistory.assignmentvector* = nil) ∧

.5 (*se.eventhistory.reactionshistory* = {})) ;

42.0 *update-history-transition* (*trom* : *Trom*, *se* : *SimulationEvent*, *ts* : *TransitionSpec*)

.1 pre (*ts.postcondition* = true)

.2 post ((*se.eventhistory.triggeredtransition* = true) ∧

.3 (*se.eventhistory.tromcurrentstate* = *trom.currentstate*) ∧

.4 (*se.eventhistory.assignmentvector* = *trom.assignmentvector*) ∧

.5 (*se.eventhistory.reactionshistory* = {})) ;

```
43.0  update-history-fire-reaction (trom : Trom, se : SimulationEvent,
 .1                                  tc : TimeConstraint, rw : ReactionWindow)
 .2  pre  ((tc.constrainedevent.label = se.eventlabel) ∧
 .3        (rw ∈ tc.reactionwindows) ∧
 .4        (se.occurtime ≥ rw.lowertimebound) ∧
 .5        (se.occurtime ≤ rw.uppertimebound))
 .6  post (∃ rh ∈ se.eventhistory.reactionshistory ·
 .7              ((rh.timeconstraint = tc) ∧
 .8               (rh.reactionwindow = rw) ∧
 .9               (rh.reaction = FIRED))) ;

44.0  update-history-disable-reaction (trom : Trom, se : SimulationEvent,
 .1                                    tc : TimeConstraint, rw : ReactionWindow)
 .2  pre  ((trom.currentstate ∈ tc.disablingstates) ∧
 .3        (rw ∈ tc.reactionwindows))
 .4  post (∃ rh ∈ se.eventhistory.reactionshistory ·
 .5              ((rh.timeconstraint = tc) ∧
 .6               (rh.reactionwindow = rw) ∧
 .7               (rh.reaction = DISABLED))) ;

45.0  update-history-enable-reaction (trom : Trom, se : SimulationEvent,
 .1                                   tc : TimeConstraint, ts : TransitionSpec)
 .2  ext rd  CLOCK' : N
 .3  pre  (ts.label = tc.transition.label)
 .4  post (let rw : ReactionWindow be st
 .5                rw = mk-ReactionWindow (tc.timebounds.lowertimebound + CLOCK',
 .6                                        tc.timebounds.uppertimebound + CLOCK') in
 .7        (∃ rh ∈ se.eventhistory.reactionshistory ·
 .8              ((rh.timeconstraint = tc) ∧
 .9               (rh.reactionwindow = rw) ∧
 .10              (rh.reaction = ENABLED)))) ;
```

```
46.0  fire-reaction (trom : Trom, se : SimulationEvent,
 .1                          tc : TimeConstraint, rw : ReactionWindow)
 .2  pre  ((tc.constrainedevent.label = se.eventlabel) ∧
 .3        (rw ∈ tc.reactionwindows) ∧
 .4        (se.occurtime ≥ rw.lowertimebound) ∧
 .5        (se.occurtime ≤ rw.uppertimebound))
 .6  post (rw ∉ tc.reactionwindows) ;

47.0  disable-reaction (trom : Trom, se : SimulationEvent,
 .1                          tc : TimeConstraint, rw : ReactionWindow)
 .2  ext rd SUBSYSTEM : Subsystem
 .3      wr SIMULATIONEVENTLIST : SimulationEvent*
 .4  pre  ((trom.currentstate ∈ tc.disablingstates) ∧
 .5        (rw ∈ tc.reactionwindows))
 .6  post ((rw ∉ tc.reactionwindows) ∧
 .7        (let se2 : SimulationEvent be st
 .8                se2 = get-enabled-simevent (trom, tc) in
 .9         (((tc.constrainedevent.type = INTERNAL) ∧
.10           (SIMULATIONEVENTLIST =
.11            [SIMULATIONEVENTLIST (i) | i ∈ inds SIMULATIONEVENTLIST ·
.12                  SIMULATIONEVENTLIST (i) ≠ se2])) ∨
.13          ((tc.constrainedevent.type = OUTPUT) ∧
.14           (let tromporttuple : [TromPortTuple] be st
.15                 tromporttuple = get-linked-tromport-tuple
.16                     (mk-TromPortTuple (se2.tromlabel, se2.portlabel),
.17                      SUBSYSTEM) in
.18            (let se3 : SimulationEvent be st
.19                  se3 = get-enabled-simevent-synch (tromporttuple, tc) in
.20             SIMULATIONEVENTLIST =
.21             [SIMULATIONEVENTLIST (i) | i ∈ inds SIMULATIONEVENTLIST ·
.22                   SIMULATIONEVENTLIST (i) ≠ se2 ∧
.23                   SIMULATIONEVENTLIST (i) ≠ se3])))))) ;
```

```
48.0  enable-reaction (trom : Trom, se : SimulationEvent,
 .1                            tc : TimeConstraint, ts : TransitionSpec)
 .2  ext rd CLOCK : N
 .3      rd SUBSYSTEM : Subsystem
 .4      wr SIMULATIONEVENTLIST : SimulationEvent*
 .5  pre (ts.label = tc.transition.label)
 .6  post (let rw : ReactionWindow be st
 .7              rw = mk-ReactionWindow (tc.timebounds.lowertimebound + CLOCK,
 .8                                      tc.timebounds.uppertimebound + CLOCK) in
 .9        (let port : Port be st
.10               port = get-lru-port (tc.constrainedevent.porttype.portlist) in
.11         (let occurtime : N be st
.12                occurtime = get-random-time-within-rw (rw) in
.13          (let se2 : SimulationEvent be st
.14                 se2 = mk-SimulationEvent
.15                       (tc.constrainedevent.label, trom.label, port.label, occurtime, nil ) in
.16           ((rw ∈ tc.reactionwindows) ∧
.17            (se2 ∈ elems SIMULATIONEVENTLIST) ∧
.18            (((tc.constrainedevent.type = OUTPUT) ∧
.19              (let tromporttuple : [TromPortTuple] be st
.20                     tromporttuple = get-linked-tromport-tuple
.21                           (mk-TromPortTuple (se2.tromlabel, se2.portlabel),
.22                            SUBSYSTEM) in
.23               (((tromporttuple ≠ nil ) ∧
.24                 (let se3 : SimulationEvent be st
.25                        se3 = mk-SimulationEvent
.26                              (se2.eventlabel, tromporttuple.tromlabel,
.27                               tromporttuple.portlabel, se2.occurtime, nil ) in
.28                  (se3 ∈ elems SIMULATIONEVENTLIST))) ∨
.29                (tromporttuple = nil )))) ∨
.30             (tc.constrainedevent.type = INTERNAL))))))) :
```

```
49.0  get-enabled-simevent (trom : Trom, tc : TimeConstraint) se : SimulationEvent
 .1   ext rd CLOCK : N
 .2       rd SIMULATIONEVENTLIST : SimulationEvent*
 .3   pre  (tc ∈ trom.timeconstraints)
 .4   post ((se ∈ elems SIMULATIONEVENTLIST) ∧
 .5         (se.eventlabel = tc.constrainedevent.label) ∧
 .6         (se.tromlabel = trom.label) ∧
 .7         (se.occurtime ≥ CLOCK) ∧
 .8         (se.eventhistory = nil )) ;

50.0  get-enabled-simevent-synch (tromporttuple : TromPortTuple,
 .1                                              tc : TimeConstraint) se : SimulationEvent
 .2   ext rd CLOCK : N
 .3       rd SIMULATIONEVENTLIST : SimulationEvent*
 .4   post ((se ∈ elems SIMULATIONEVENTLIST) ∧
 .5         (se.eventlabel = tc.constrainedevent.label) ∧
 .6         (se.tromlabel = tromporttuple.tromlabel) ∧
 .7         (se.occurtime ≥ CLOCK) ∧
 .8         (se.eventhistory = nil )) ;
```

```
51.0  schedule-unconstrained-internal-events-from-initial-state ()
 .1   ext rd CLOCK : N
 .2       rd SUBSYSTEM : Subsystem
 .3       wr SIMULATIONEVENTLIST : SimulationEvent*
 .4   pre  CLOCK = 0
 .5   post ((CLOCK = 0) ∧
 .6         (∀ trom ∈ {trom | trom : Trom · exists-in-subsystem (trom, SUBSYSTEM)} ·
 .7              (let event : [Event] = get-unconstrained-internal-event (trom) in
 .8               (((event ≠ nil ) ∧
 .9                 (let se : SimulationEvent be st
 .10                          se = mk-SimulationEvent
 .11                               (event.label, trom.label, "NULLPORT", CLOCK, nil ) in
 .12                  ((se ∈ elems SIMULATIONEVENTLIST) ∧
 .13                   (let i : N1 be st
 .14                            SIMULATIONEVENTLIST (i) = se in
 .15                    (∀ se2 ∈ elems SIMULATIONEVENTLIST ·
 .16                         (let j : N1 be st
 .17                                  SIMULATIONEVENTLIST (j) = se2 in
 .18                          i < j)))))) ∨
 .19               (event = nil ))))) :
```

```
52.0  schedule-unconstrained-internal-event (trom : Trom, se : SimulationEvent)
 .1   ext rd CLOCK : N
 .2       wr SIMULATIONEVENTLIST : SimulationEvent*
 .3   pre  ((se ∈ elems SIMULATIONEVENTLIST) ∧ (se.tromlabel = trom.label))
 .4   post (let event : [Event] = get-unconstrained-internal-event (trom) in
 .5         (((event ≠ nil ) ∧
 .6           (let j : N1 be st
 .7                   j = get-simevent-index (se, SIMULATIONEVENTLIST) in
 .8            (let se2 : SimulationEvent be st
 .9                    se2 = mk-SimulationEvent
 .10                        (event.label, trom.label, "NULLPORT", CLOCK, nil ) in
 .11            (SIMULATIONEVENTLIST =
 .12             [SIMULATIONEVENTLIST~ (i) |
 .13                  i ∈ inds SIMULATIONEVENTLIST~ · i ≤ j] ⌢
 .14             [se2] ⌢
 .15             [SIMULATIONEVENTLIST~ (i) |
 .16                  i ∈ inds SIMULATIONEVENTLIST~ · i > j])))) ∨
 .17         (event = nil ))) ;

53.0  initialize-simulation-clock ()
 .1   ext wr CLOCK~ : N
 .2   post CLOCK~ = 0 ;

54.0  update-simulation-clock ()
 .1   ext wr CLOCK~ : N
 .2   post CLOCK~ = CLOCK~ + 1

end TROM
```

B.2 Specification of Simulator Using Explicit Operations

module *TROM*

exports all

definitions

types

functions

1.0 $get\text{-}trom\text{-}object : String \times Subsystem \rightarrow [Trom]$

.1 $get\text{-}trom\text{-}object\,(tromlabel, subsystem) \triangleq$

.2 (let $trom : [Trom]$ be st

.3 $((trom \in subsystem.troms \wedge trom.label = tromlabel) \vee$

.4 $(\exists!\, s \in subsystem.includes \cdot$

.5 $(trom = get\text{-}trom\text{-}object\,(tromlabel, s))) \vee$

.6 $(trom = \text{nil}\ ))$ in

.7 $trom$);

2.0 $get\text{-}transition\text{-}spec : Trom \times SimulationEvent \rightarrow [TransitionSpec]$

.1 $get\text{-}transition\text{-}spec\,(trom, se) \triangleq$

.2 (let $ts : [TransitionSpec]$ be st

.3 $(((ts \in trom.transitionspecs) \wedge$

.4 $((trom.currentstate.label = ts.sourcestate.label) \vee$

.5 $(substate\text{-}of\,(trom.currentstate, ts.sourcestate))) \wedge$

.6 $(ts.triggerevent.label = se.eventlabel) \wedge$

.7 $(ts.portcondition = \text{true}) \wedge$

.8 $(ts.enabcondition = \text{true})) \vee$

.9 $(ts = \text{nil}\ ))$ in

.10 ts)

.11 pre $(trom.label = se.tromlabel)$;

3.0 $substate\text{-}of : State \times State \rightarrow \mathbb{B}$

.1 $substate\text{-}of\,(substate, complexstate) \triangleq$

.2 $(substate \in complexstate.substates \vee$

.3 $(\exists!\, s \in complexstate.substates \cdot$

.4 $(s.type = \text{COMPLEX} \wedge substate\text{-}of\,(substate, s))))$

.5 pre $(complexstate.type = \text{COMPLEX})$:

```
4.0  get-entry-state : State → State
 .1  get-entry-state (complexstate) ≜
 .2    (let entry : State be st
 .3          (∃! s ∈ complexstate.substates ·
 .4               ((s.isinitial = true) ∧
 .5                ((s.type = SIMPLE ∧ entry = s) ∨
 .6                 (s.type = COMPLEX ∧ entry = get-entry-state (s))))) in
 .7     entry)
 .8  pre (complexstate.type = COMPLEX) ;

5.0  get-initial-state : Trom → State
 .1  get-initial-state (trom) ≜
 .2    (let initial : State be st
 .3          (∃! s ∈ trom.states ·
 .4               ((s.isinitial = true) ∧
 .5                ((s.type = SIMPLE ∧ initial = s) ∨
 .6                 (s.type = COMPLEX ∧ initial = get-entry-state (s))))) in
 .7     initial)
 .8  pre (trom.states ≠ {}) ;

6.0  get-linked-tromport-tuple : TromPortTuple × Subsystem → [TromPortTuple]
 .1  get-linked-tromport-tuple (tupleA, subsystem) ≜
 .2    (let tupleB : [TromPortTuple] be st
 .3          ((∃! pl ∈ subsystem.portlinks ·
 .4               ((pl.tromporttuple1 = tupleA ∧ pl.tromporttuple2 = tupleB) ∨
 .5                (pl.tromporttuple2 = tupleA ∧ pl.tromporttuple1 = tupleB))) ∨
 .6           (∃! s ∈ subsystem.includes ·
 .7                (tupleB = get-linked-tromport-tuple (tupleA, s))) ∨
 .8           (tupleB = undefined )) in
 .9     tupleB);

7.0  exists-in-subsystem : Trom × Subsystem → B
 .1  exists-in-subsystem (trom, subsys) ≜
 .2    (trom ∈ subsys.troms ∨
 .3     (∃! subsystem ∈ subsys.includes ·
 .4          (exists-in-subsystem (trom, subsystem))))
 .5  pre ((subsys.troms ≠ {}) ∨ (subsys.includes ≠ {})) ;
```

```
8.0  get-unconstrained-internal-event : Trom → [Event]
 .1  get-unconstrained-internal-event (trom) ≜
 .2    (let event : [Event] be st
 .3            ((∃ ts ∈ trom.transitionspecs ·
 .4                  ((ts.sourcestate = trom.currentstate) ∧
 .5                   (ts.triggerevent.type = INTERNAL) ∧
 .6                   (¬ constrained-event (trom, ts.triggerevent)) ∧
 .7                   (event = ts.triggerevent))) ∨
 .8             (event = undefined )) in
 .9     event):

9.0  constrained-event : Trom × Event → B
 .1  constrained-event (trom, event) ≜
 .2    (∃ tc ∈ trom.timeconstraints ·
 .3          (tc.constrainedevent = event))
 .4  pre (event ∈ trom.events) ;

10.0  get-simevent-index : SimulationEvent × SimulationEvent* → N1
 .1  get-simevent-index (se, se-list) ≜
 .2    (let index : N1 be st
 .3            (se-list (index) = se) in
 .4     index)
 .5  pre (se ∈ elems se-list) ;

11.0  get-random-time-within-rw : ReactionWindow → N
 .1  get-random-time-within-rw (rw) ≜
 .2    (let time : N be st
 .3            ((time ≥ rw.lowertimebound) ∧ (time ≤ rw.uppertimebound)) in
 .4     time);

12.0  get-lru-port : Port* → Port
 .1  get-lru-port (portlist) ≜
 .2    (let port : Port be st
 .3            (port ∈ elems portlist) in
 .4     port)
 .5  pre (portlist ≠ [])
```

operations

```
13.0  simulator : () ==o> ()
 .1   simulator() ≜
 .2     (dcl i : N1 := 1;
 .3      initialize-simulation-clock() ;
 .4      schedule-unconstrained-internal-events-from-initial-state() ;
 .5      while (i ≤ len SIMULATIONEVENTLIST)
 .6      do (while (CLOCK < SIMULATIONEVENTLIST(i).occurtime)
 .7          do (update-simulation-clock() );
 .8          while ((i ≤ len SIMULATIONEVENTLIST) ∧
 .9                 (CLOCK = SIMULATIONEVENTLIST(i).occurtime))
 .10         do (handle-event(SIMULATIONEVENTLIST(i)) ;
 .11             i := i + 1)))
 .12   pre ((SIMULATIONEVENTLIST ≠ []) ∧
 .13       (∀ se ∈ elems SIMULATIONEVENTLIST ·
 .14            ((se.occurtime ≥ CLOCK) ∧
 .15             (se.eventhistory = nil ))) ∧
 .16       (∀ trom ∈ {trom | trom : Trom · exists-in-subsystem(trom, SUBSYSTEM)} ·
 .17            ((trom.currentstate = get-initial-state(trom)) ∧
 .18             (∀ tc ∈ trom.timeconstraints ·
 .19                  (tc.reactionwindows = {})))))
 .20    post ((SIMULATIONEVENTLIST ≠ []) ∧
 .21         (SIMULATIONEVENTLIST(len SIMULATIONEVENTLIST).occurtime =
 .22          CLOCK) ∧
 .23         (∀ se ∈ elems SIMULATIONEVENTLIST ·
 .24              ((se.occurtime ≤ CLOCK) ∧
 .25               (se.eventhistory ≠ nil ))) ∧
 .26         (∀ trom ∈ {trom | trom : Trom · exists-in-subsystem(trom, SUBSYSTEM)} ·
 .27              (∀ tc ∈ trom.timeconstraints ·
 .28                   (tc.reactionwindows = {}))))) ;
```

```
14.0  handle-event : SimulationEvent ⟶° ()
 .1   handle-event (se) ≙
 .2     (dcl trom : [Trom],
 .3          ts : [TransitionSpec];
 .4      trom := get-trom-object (se.tromlabel, SUBSYSTEM);
 .5      if trom = nil
 .6      then return
 .7      else skip;
 .8      ts := get-transition-spec (trom, se);
 .9      if ts = nil
 .10     then (update-history-notransition(trom, se, ts) )
 .11     else (update-history-assignment-vector(trom, se, ts) :
 .12          if ts.postcondition = false
 .13          then (update-history-notransition(trom, se, ts) )
 .14          else (update-history-transition(trom, se, ts) ;
 .15               update-trom-current-state(trom, se, ts) ;
 .16               handle-transition(trom, se, ts) ;
 .17               schedule-unconstrained-internal-event(trom, se) )))
 .18 pre (se.occurtime = CLOCK)
 .19   post (CLOCK = ~CLOCK) :
```

```
15.0  handle-transition : Trom × SimulationEvent × TransitionSpec ⟶° ()
 .1   handle-transition (trom, se, ts) ≙
 .2     (for all tc ∈ trom.timeconstraints
 .3      do (if tc.constrainedevent.label = se.eventlabel
 .4          then (for all rw ∈ tc.reactionwindows
 .5                do (if se.occurtime ≥ rw.lowertimebound ∧
 .6                       se.occurtime ≤ rw.uppertimebound
 .7                    then (update-history-fire-reaction(trom, se, tc, rw) ;
 .8                          fire-reaction(trom, se, tc, rw) )
 .9                    else skip))
.10          else skip;
.11          if trom.currentstate ∈ tc.disablingstates
.12          then (for all rw ∈ tc.reactionwindows
.13                do (update-history-disable-reaction(trom, se, tc, rw) ;
.14                    disable-reaction(trom, se, tc, rw) ))
.15          else skip;
.16          if ts.label = tc.transition.label
.17          then (update-history-enable-reaction(trom, se, tc, ts) ;
.18                enable-reaction(trom, se, tc, ts) )
.19          else skip))
.20  pre (se.occurtime = CLOCK)
.21    post (CLOCK = \overline{CLOCK}) ;
```

16.0 *update-trom-current-state* (*trom* : *Trom*, *se* : *SimulationEvent*, *ts* : *TransitionSpec*)

.1 pre $(ts.postcondition = \text{true})$

.2 post $(((ts.destinstate.type = \text{SIMPLE}) \wedge$

.3 $(trom.currentstate = ts.destinstate)) \vee$

.4 $((ts.destinstate.type = \text{COMPLEX}) \wedge$

.5 $(trom.currentstate = get\text{-}entry\text{-}state\,(ts.destinstate))))$;

17.0 *update-history-assignment-vector* (*trom* : *Trom*, *se* : *SimulationEvent*, *ts* : [*TransitionSpec*])

.1 pre $(ts \neq \text{nil})$

.2 post $(se.eventhistory.assignmentvector = trom.assignmentvector)$;

18.0 *update-history-notransition* (*trom* : *Trom*, *se* : *SimulationEvent*, *ts* : [*TransitionSpec*])

.1 pre $((ts = \text{nil}) \vee (ts.postcondition = \text{false}))$

.2 post $((se.eventhistory.triggeredtransition = \text{false}) \wedge$

.3 $(se.eventhistory.tromcurrentstate = \text{nil}) \wedge$

.4 $(se.eventhistory.assignmentvector = \text{nil}) \wedge$

.5 $(se.eventhistory.reactionshistory = \{\}))$;

19.0 *update-history-transition* (*trom* : *Trom*, *se* : *SimulationEvent*, *ts* : *TransitionSpec*)

.1 pre $(ts.postcondition = \text{true})$

.2 post $((se.eventhistory.triggeredtransition = \text{true}) \wedge$

.3 $(se.eventhistory.tromcurrentstate = trom.currentstate) \wedge$

.4 $(se.eventhistory.assignmentvector = trom.assignmentvector) \wedge$

.5 $(se.eventhistory.reactionshistory = \{\}))$;

```
20.0  update-history-fire-reaction :
 .1                  Trom × SimulationEvent × TimeConstraint × ReactionWindow →° ()
 .2  update-history-fire-reaction (trom, se, tc, rw) ≜
 .3    insert-rhistory(se, mk-ReactionHistory (tc, rw, FIRED))
 .4  pre ((tc.constrainedevent.label = se.eventlabel) ∧
 .5       (rw ∈ tc.reactionwindows) ∧
 .6       (se.occurtime ≥ rw.lowertimebound) ∧
 .7       (se.occurtime ≤ rw.uppertimebound))
 .8    post (let rh = mk-ReactionHistory (tc, rw, FIRED) in
 .9          (rh ∈ se.eventhistory.reactionshistory)) :

21.0  update-history-disable-reaction :
 .1                  Trom × SimulationEvent × TimeConstraint × ReactionWindow →° ()
 .2  update-history-disable-reaction (trom, se, tc, rw) ≜
 .3    insert-rhistory(se, mk-ReactionHistory (tc, rw, DISABLED))
 .4  pre ((trom.currentstate ∈ tc.disablingstates) ∧
 .5       (rw ∈ tc.reactionwindows))
 .6    post (let rh = mk-ReactionHistory (tc, rw, DISABLED) in
 .7          (rh ∈ se.eventhistory.reactionshistory)) :

22.0  update-history-enable-reaction :
 .1                  Trom × SimulationEvent × TimeConstraint × TransitionSpec →° ()
 .2  update-history-enable-reaction (trom, se, tc, ts) ≜
 .3    (dcl rw : ReactionWindow;
 .4     rw := mk-ReactionWindow (tc.timebounds.lowertimebound + CLOCK,
 .5                              tc.timebounds.uppertimebound + CLOCK);
 .6     insert-rhistory(se, mk-ReactionHistory (tc, rw, ENABLED)) )
 .7  pre (ts.label = tc.transition.label)
 .8    post (let rw = mk-ReactionWindow (tc.timebounds.lowertimebound + CLOCK,
 .9                                      tc.timebounds.uppertimebound + CLOCK) in
.10          (let rh = mk-ReactionHistory (tc, rw, ENABLED) in
.11           (rh ∈ se.eventhistory.reactionshistory))) ;
```

```
23.0  fire-reaction : Trom × SimulationEvent × TimeConstraint × ReactionWindow ⟶° ()
 .1   fire-reaction (trom, se, tc, rw) ≜
 .2     remove-rwindow(tc, rw)
 .3   pre ((tc.constrainedevent.label = se.eventlabel) ∧
 .4        (rw ∈ tc.reactionwindows) ∧
 .5        (se.occurtime ≥ rw.lowertimebound) ∧
 .6        (se.occurtime ≤ rw.uppertimebound))
 .7     post (rw ∉ tc.reactionwindows) ;

24.0  disable-reaction : Trom × SimulationEvent × TimeConstraint × ReactionWindow ⟶° ()
 .1   disable-reaction (trom, se, tc, rw) ≜
 .2     (dcl se2 : SimulationEvent,
 .3          se3 : SimulationEvent,
 .4          tromporttuple : [TromPortTuple];
 .5      remove-rwindow(tc, rw) ;
 .6      se2 := get-enabled-simevent (trom, tc);
 .7      if tc.constrainedevent.type = OUTPUT
 .8      then (tromporttuple := get-linked-tromport-tuple
 .9                                  (mk-TromPortTuple (se2.tromlabel, se2.portlabel),
.10                                   SUBSYSTEM);
.11            se3 := get-enabled-simevent-synch (tromporttuple, tc);
.12            remove-simevent(se3) )
.13      else skip:
.14      remove-simevent(se2) )
.15   pre ((trom.currentstate ∈ tc.disablingstates) ∧
.16        (rw ∈ tc.reactionwindows))
.17     post (rw ∉ tc.reactionwindows) ;
```

```
25.0  enable-reaction : Trom × SimulationEvent × TimeConstraint × TransitionSpec ~o~> ()
 .1   enable-reaction (trom, se, tc, ts) ≜
 .2     (dcl rw : ReactionWindow;
 .3      rw := mk-ReactionWindow (tc.timebounds.lowertimebound + CLOCK,
 .4                               tc.timebounds.uppertimebound + CLOCK);
 .5      insert-rwindow(tc, rw) ;
 .6      schedule-enabled-reaction(trom, tc, ts, rw) )
 .7   pre (ts.label = tc.transition.label)
 .8     post (let rw : ReactionWindow be st
 .9                 rw = mk-ReactionWindow (tc.timebounds.lowertimebound + CLOCK,
.10                                        tc.timebounds.uppertimebound + CLOCK) in
.11           rw ∈ tc.reactionwindows) ;

26.0  schedule-enabled-reaction :
 .1                 Trom × TimeConstraint × TransitionSpec × ReactionWindow ~o~> ()
 .2   schedule-enabled-reaction (trom, tc, ts, rw) ≜
 .3     (dcl port : Port,
 .4          occurtime : N,
 .5          se2 : SimulationEvent,
 .6          se3 : SimulationEvent,
 .7          tromporttuple : [TromPortTuple];
 .8      port := get-lru-port (tc.constrainedevent.porttype.portlist);
 .9      occurtime := get-random-time-within-rw (rw);
.10      se2 := mk-SimulationEvent
.11                (tc.constrainedevent.label, trom.label, port.label, occurtime, nil );
.12      insert-simevent(se2) ;
.13      if tc.constrainedevent.type = OUTPUT
.14      then (tromporttuple := get-linked-tromport-tuple
.15                              (mk-TromPortTuple (se2.tromlabel, se2.portlabel),
.16                               SUBSYSTEM);
.17            if tromporttuple ≠ nil
.18            then (se3 := mk-SimulationEvent
.19                         (se2.eventlabel, tromporttuple.tromlabel,
.20                          tromporttuple.portlabel, se2.occurtime, nil );
.21                  insert-simevent(se3) )
.22            else skip)
.23      else skip)
.24   pre ((ts.label = tc.transition.label) ∧ (rw ∈ tc.reactionwindows)) ;
```

```
27.0  get-enabled-simevent (trom : Trom, tc : TimeConstraint) se : SimulationEvent
 .1   ext rd CLOCK : N
 .2        rd SIMULATIONEVENTLIST : SimulationEvent*
 .3   pre  (tc ∈ trom.timeconstraints)
 .4   post ((se ∈ elems SIMULATIONEVENTLIST) ∧
 .5         (se.eventlabel = tc.constrainedevent.label) ∧
 .6         (se.tromlabel = trom.label) ∧
 .7         (se.occurtime ≥ CLOCK) ∧
 .8         (se.eventhistory = nil )) ;

28.0  get-enabled-simevent-synch (tromporttuple : TromPortTuple,
 .1                                          tc : TimeConstraint) se : SimulationEvent
 .2   ext rd CLOCK : N
 .3        rd SIMULATIONEVENTLIST : SimulationEvent*
 .4   post ((se ∈ elems SIMULATIONEVENTLIST) ∧
 .5         (se.eventlabel = tc.constrainedevent.label) ∧
 .6         (se.tromlabel = tromporttuple.tromlabel) ∧
 .7         (se.occurtime ≥ CLOCK) ∧
 .8         (se.eventhistory = nil )) ;

29.0  schedule-unconstrained-internal-events-from-initial-state : () →° ()
 .1   schedule-unconstrained-internal-events-from-initial-state () ≜
 .2     (dcl event : [Event],
 .3          se : SimulationEvent;
 .4      for all trom ∈ {trom | trom : Trom · exists-in-subsystem (trom, SUBSYSTEM)}
 .5      do (event := get-unconstrained-internal-event (trom);
 .6          if event ≠ nil
 .7          then (se := mk-SimulationEvent (event.label, trom.label,
 .8                                          "NULLPORT", CLOCK, nil );
 .9                SIMULATIONEVENTLIST := [se]⌢ SIMULATIONEVENTLIST)
 .10         else skip))
 .11  pre  (CLOCK = 0)
 .12    post (CLOCK = 0) ;
```

```
30.0  schedule-unconstrained-internal-event : Trom × SimulationEvent ⟶° ()
 .1   schedule-unconstrained-internal-event (trom, se) ≜
 .2     (dcl event : [Event],
 .3          se2 : SimulationEvent,
 .4          j : N1;
 .5      event := get-unconstrained-internal-event (trom);
 .6      if event ≠ nil
 .7      then (se2 := mk-SimulationEvent (event.label, trom.label,
 .8                                       "NULLPORT", CLOCK, nil );
 .9            j := get-simevent-index (se, SIMULATIONEVENTLIST);
 .10           SIMULATIONEVENTLIST :=
 .11               [SIMULATIONEVENTLIST (i) |
 .12                    i ∈ inds SIMULATIONEVENTLIST · i ≤ j] ⌢
 .13               [se2] ⌢
 .14               [SIMULATIONEVENTLIST (i) |
 .15                    i ∈ inds SIMULATIONEVENTLIST · i > j])
 .16     else skip)
 .17  pre ((se ∈ elems SIMULATIONEVENTLIST) ∧ (se.tromlabel = trom.label))
 .18    post (CLOCK = CLOCK~) :

31.0  insert-simevent : SimulationEvent ⟶° ()
 .1   insert-simevent (se) ≜
 .2     SIMULATIONEVENTLIST :=
 .3         [SIMULATIONEVENTLIST (i) |
 .4              i ∈ inds SIMULATIONEVENTLIST ·
 .5                   SIMULATIONEVENTLIST (i).occurtime ≤ se.occurtime] ⌢
 .6         [se] ⌢
 .7         [SIMULATIONEVENTLIST (i) |
 .8              i ∈ inds SIMULATIONEVENTLIST ·
 .9                   SIMULATIONEVENTLIST (i).occurtime > se.occurtime]
 .10  pre (se ∉ elems SIMULATIONEVENTLIST)
 .11    post (se ∈ elems SIMULATIONEVENTLIST) ;
```

```
32.0  remove-simevent : SimulationEvent ⟶° ()
 .1   remove-simevent (se) ≜
 .2     (dcl j : N1;
 .3      j := get-simevent-index (se, SIMULATIONEVENTLIST);
 .4      SIMULATIONEVENTLIST :=
 .5           [SIMULATIONEVENTLIST (i) |
 .6                i ∈ inds SIMULATIONEVENTLIST · i < j] ⌢
 .7           [SIMULATIONEVENTLIST (i) |
 .8                i ∈ inds SIMULATIONEVENTLIST · i > j])
 .9   pre  (se ∈ elems SIMULATIONEVENTLIST)
 .10    post (se ∉ elems SIMULATIONEVENTLIST) ;

33.0  insert-rwindow (tc : TimeConstraint, rw : ReactionWindow)
 .1   pre  (rw ∉ tc.reactionwindows)
 .2   post (tc.reactionwindows = tc.reactionwindows ∪ {rw}) ;

34.0  remove-rwindow (tc : TimeConstraint, rw : ReactionWindow)
 .1   pre  (rw ∈ tc.reactionwindows)
 .2   post (tc.reactionwindows = tc.reactionwindows \ {rw}) ;

35.0  insert-rhistory (se : SimulationEvent, rh : ReactionHistory)
 .1   pre  (rh ∉ se.eventhistory.reactionshistory)
 .2   post (se.eventhistory.reactionshistory = se.eventhistory.reactionshistory ∪ {rh}) ;

36.0  remove-rhistory (se : SimulationEvent, rh : ReactionHistory)
 .1   pre  (rh ∈ se.eventhistory.reactionshistory)
 .2   post (se.eventhistory.reactionshistory = se.eventhistory.reactionshistory \ {rh}) ;

37.0  initialize-simulation-clock : () ⟶° ()
 .1   initialize-simulation-clock () ≜
 .2     CLOCK := 0;

38.0  update-simulation-clock : () ⟶° ()
 .1   update-simulation-clock () ≜
 .2     CLOCK := CLOCK + 1

end TROM
```

Appendix C

Simulation Example

The animation tool has been exercised to simulate the Train-Gate-Controller system. The input includes the description of the LSL trait Set, the TROM *class and system specifications, and the initial list of simulation events. The output includes the textual description of the status of the system and the simulation event list. Whenever an event is handled, the display includes a detailed description of the operation representing the computational step. We include the input and output of the simulation run in sections C.1 and C.2 respectively.*

C.1 Simulation Input for Train-Gate-Controller System

```
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 [@C]
   Events: Near!C, Exit!C, In, Out
   States: *S1, S2, S3, S4
   Attributes: cr:@C
   Attribute-function: S1 -> {}; S2 -> {cr}; S3 -> {}; S4 -> {};
   Transition-Spec:
      R1: <S1,S2>; Near(true); true => cr' = pid;
      R2: <S2,S3>; In; true => true;
      R3: <S3,S4>; Out; true => true;
      R4: <S4,S1>; Exit(pid = cr); true => true;
   Time-Constraints:
      TC1: (R1, In, [8,16], {});
      TC2: (R2, Exit, [0,8], {});
end

Class Controller [@P, @G]
   Events: Near?P, Exit?P, Lower!G, Raise!G
   States: *C1, C2, C3, C4
   Attributes: inSet:PSet
   Traits: Set[@P, PSet]
   Attribute-function: C1 -> {}; C2 -> {inSet}; C3 -> {inSet}; C4 -> {inSet};
   Transition-Spec:
      R1: <C1,C2>; Near(true); true => inSet' = insert(pid, inSet);
      R2: <C2,C2>,<C3,C3>; Near(NOT(belongto(pid, inSet)));
                                   true => inSet' = 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' = delete(pid, inSet);
      R6: <C4,C1>; Raise(true); true => true;
   Time-Constraints:
      TC1: (R1, Lower, [0,4], {});
      TC2: (R5, Raise, [0,4], {});
end

Class Gate [@S]
   Events: Lower?S, Raise?S, Down, Up
   States: *G1, G2, G3, G4
   Transition-Spec:
      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, [0,4], {});
      TC2: (R3, Up, [4,8], {});
end

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

```
      t3.@C2 <-> c2.@P3;
      c1.@G1 <-> g1.@S1;
      c2.@G1 <-> g2.@S1;
end

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

C.2 Simulation Output for Train-Gate-Controller System

```
Starting TROMLAB Environment.

Type Checking Specifications. Please wait...

Specifications type-checked.

     Start simulation? (y/n): y
     Please enter system label: TCG
Starting Simulation of Subsystem TCG

     Set debugger mode? (y/n): n
     Normal/increased/decreased clock pace (n/i/d): n
     Set time-out period (time units - default 20): 20

 Scheduling Sim    Event: Near    Trom: c1    Port: P1     Time: 3
  History:
  Event Outcome: NOTYET_HANDLED
 Scheduling Sim    Event: Near    Trom: c2    Port: P2     Time: 5
  History:
  Event Outcome: NOTYET_HANDLED
 Scheduling Sim    Event: Near    Trom: c1    Port: P3     Time: 7
  History:
  Event Outcome: NOTYET_HANDLED

Current Simulation Time: 0
Subsystem status:
 Subsystem label: TCG
 Trom Status:   Trom-label: t1    Trom-class: Train    Current-state: S1
  AssignmentVector:
   Attribute label: cr    Attribute type: PORT_TYPE
    Port-ID: NULL
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Trom Status:   Trom-label: t2    Trom-class: Train    Current-state: S1
  AssignmentVector:
   Attribute label: cr    Attribute type: PORT_TYPE
```

```
    Port-ID: NULL
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Trom Status:   Trom-label: t3    Trom-class: Train    Current-state: S1
  AssignmentVector:
   Attribute label: cr    Attribute type: PORT_TYPE
    Port-ID: NULL
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Trom Status:   Trom-label: c1    Trom-class: Controller    Current-state: C1
  AssignmentVector:
   Attribute label: inSet    Attribute type: TRAIT_TYPE
   Trait type: PSet        Trait name: Set
   Trait value: NULL-SET
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Trom Status:   Trom-label: c2    Trom-class: Controller    Current-state: C1
  AssignmentVector:
   Attribute label: inSet    Attribute type: TRAIT_TYPE
   Trait type: PSet        Trait name: Set
   Trait value: NULL-SET
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Trom Status:   Trom-label: g1    Trom-class: Gate    Current-state: G1
  AssignmentVector: NULL-VECTOR
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Trom Status:   Trom-label: g2    Trom-class: Gate    Current-state: G1
  AssignmentVector: NULL-VECTOR
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
Subsystem TCG is in stable state.
```

```
Simulation Event List:
 Sim-Event 1:    Event: Near    Trom: t1    Port: C1     Time: 3
  History:
  Event Outcome: NOTYET_HANDLED
 Sim-Event 2:    Event: Near    Trom: c1    Port: P1     Time: 3
  History:
  Event Outcome: NOTYET_HANDLED
 Sim-Event 3:    Event: Near    Trom: t2    Port: C2     Time: 5
  History:
  Event Outcome: NOTYET_HANDLED
 Sim-Event 4:    Event: Near    Trom: c2    Port: P2     Time: 5
  History:
  Event Outcome: NOTYET_HANDLED
 Sim-Event 5:    Event: Near    Trom: t3    Port: C1     Time: 7
  History:
  Event Outcome: NOTYET_HANDLED
 Sim-Event 6:    Event: Near    Trom: c1    Port: P3     Time: 7
  History:
  Event Outcome: NOTYET_HANDLED
End of Simulation Event List.

Current Simulation Time: 1
Current Simulation Time: 2
Current Simulation Time: 3
Handling    Sim    Event: Near    Trom: t1    Port: C1     Time: 3
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: t1    Trom-class: Train    Current-state: S1
  AssignmentVector:
   Attribute label: cr    Attribute type: PORT_TYPE
    Port-ID: NULL
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
Triggering Transition:
 Label: R1    Source: S1    Destin: S2    Trigger Event: Near
Enabling Reaction - Time Constraint:
 Label: TC1    Transition: R1    Constrained Event: In    Bounds: [8,16]
Scheduling Sim    Event: In    Trom: t1    Port: NULLPORT     Time: 19
```

```
  History:
  Event Outcome: NOTYET_HANDLED

Current Simulation Time: 3
Handling    Sim    Event: Near    Trom: c1    Port: P1    Time: 3
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: c1    Trom-class: Controller    Current-state: C1
  AssignmentVector:
   Attribute label: inSet    Attribute type: TRAIT_TYPE
   Trait type: PSet        Trait name: Set
   Trait value: NULL-SET
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Triggering Transition:
  Label: R1    Source: C1    Destin: C2    Trigger Event: Near
 Enabling Reaction - Time Constraint:
  Label: TC1    Transition: R1    Constrained Event: Lower    Bounds: [0,4]
 Scheduling Sim    Event: Lower    Trom: c1    Port: G1    Time: 3
  History:
  Event Outcome: NOTYET_HANDLED
 Scheduling Sim    Event: Lower    Trom: g1    Port: S1    Time: 3
  History:
  Event Outcome: NOTYET_HANDLED

Current Simulation Time: 3
Handling    Sim    Event: Lower    Trom: c1    Port: G1    Time: 3
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: c1    Trom-class: Controller    Current-state: C2
  AssignmentVector:
   Attribute label: inSet    Attribute type: TRAIT_TYPE
   Trait type: PSet        Trait name: Set
   Trait value:
    Port-ID: P1
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : [3,7]
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
```

```
 Triggering Transition:
  Label: R3    Source: C2    Destin: C3    Trigger Event: Lower
 Firing Reaction - Time Constraint:
  Label: TC1    Transition: R1    Constrained Event: Lower    Bounds: [0,4]

Current Simulation Time: 3
Handling    Sim    Event: Lower    Trom: g1    Port: S1    Time: 3
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: g1    Trom-class: Gate    Current-state: G1
  AssignmentVector: NULL-VECTOR
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Triggering Transition:
  Label: R1    Source: G1    Destin: G2    Trigger Event: Lower
 Enabling Reaction - Time Constraint:
  Label: TC1    Transition: R1    Constrained Event: Down    Bounds: [0,4]
 Scheduling Sim    Event: Down    Trom: g1    Port: NULLPORT    Time: 3
  History:
  Event Outcome: NOTYET_HANDLED

Current Simulation Time: 3
Handling    Sim    Event: Down    Trom: g1    Port: NULLPORT    Time: 3
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: g1    Trom-class: Gate    Current-state: G2
  AssignmentVector: NULL-VECTOR
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : [3,7]
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Triggering Transition:
  Label: R2    Source: G2    Destin: G3    Trigger Event: Down
 Firing Reaction - Time Constraint:
  Label: TC1    Transition: R1    Constrained Event: Down    Bounds: [0,4]

Current Simulation Time: 3
Current Simulation Time: 4
Current Simulation Time: 5
```

```
Handling    Sim    Event: Near    Trom: t2    Port: C2    Time: 5
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: t2    Trom-class: Train    Current-state: S1
  AssignmentVector:
   Attribute label: cr    Attribute type: PORT_TYPE
    Port-ID: NULL
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Triggering Transition:
  Label: R1    Source: S1    Destin: S2    Trigger Event: Near
 Enabling Reaction - Time Constraint:
  Label: TC1    Transition: R1    Constrained Event: In    Bounds: [8,16]
 Scheduling Sim    Event: In    Trom: t2    Port: NULLPORT    Time: 17
  History:
  Event Outcome: NOTYET_HANDLED

Current Simulation Time: 5
Handling    Sim    Event: Near    Trom: c2    Port: P2    Time: 5
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: c2    Trom-class: Controller    Current-state: C1
  AssignmentVector:
   Attribute label: inSet    Attribute type: TRAIT_TYPE
   Trait type: PSet        Trait name: Set
   Trait value: NULL-SET
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Triggering Transition:
  Label: R1    Source: C1    Destin: C2    Trigger Event: Near
 Enabling Reaction - Time Constraint:
  Label: TC1    Transition: R1    Constrained Event: Lower    Bounds: [0,4]
 Scheduling Sim    Event: Lower    Trom: c2    Port: G1    Time: 5
  History:
  Event Outcome: NOTYET_HANDLED
 Scheduling Sim    Event: Lower    Trom: g2    Port: S1    Time: 5
  History:
```

```
   Event Outcome: NOTYET_HANDLED

 Current Simulation Time: 5
 Handling    Sim    Event: Lower    Trom: c2    Port: G1    Time: 5
   History:
   Event Outcome: NOTYET_HANDLED
  Active Trom:  Trom-label: c2    Trom-class: Controller    Current-state: C2
   AssignmentVector:
    Attribute label: inSet    Attribute type: TRAIT_TYPE
    Trait type: PSet        Trait name: Set
    Trait value:
     Port-ID: P2
   ReactionVector:
    ReactionSubVector: TimeConstraint: TC1 : [5,9]
    ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
  Triggering Transition:
   Label: R3    Source: C2    Destin: C3    Trigger Event: Lower
  Firing Reaction - Time Constraint:
   Label: TC1    Transition: R1    Constrained Event: Lower    Bounds: [0,4]

Current Simulation Time: 5
Handling    Sim    Event: Lower    Trom: g2    Port: S1    Time: 5
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: g2    Trom-class: Gate    Current-state: G1
  AssignmentVector: NULL-VECTOR
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Triggering Transition:
  Label: R1    Source: G1    Destin: G2    Trigger Event: Lower
 Enabling Reaction - Time Constraint:
  Label: TC1    Transition: R1    Constrained Event: Down    Bounds: [0,4]
 Scheduling Sim    Event: Down    Trom: g2    Port: NULLPORT    Time: 5
  History:
  Event Outcome: NOTYET_HANDLED

Current Simulation Time: 5
Handling    Sim    Event: Down    Trom: g2    Port: NULLPORT    Time: 5
```

```
   History:
   Event Outcome: NOTYET_HANDLED
  Active Trom:  Trom-label: g2     Trom-class: Gate     Current-state: G2
   AssignmentVector: NULL-VECTOR
   ReactionVector:
    ReactionSubVector: TimeConstraint: TC1 : [5,9]
    ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
  Triggering Transition:
   Label: R2     Source: G2     Destin: G3     Trigger Event: Down
  Firing Reaction - Time Constraint:
   Label: TC1     Transition: R1     Constrained Event: Down     Bounds: [0,4]

 Current Simulation Time: 5
 Current Simulation Time: 6
 Current Simulation Time: 7
 Handling     Sim     Event: Near     Trom: t3     Port: C1      Time: 7
   History:
   Event Outcome: NOTYET_HANDLED
  Active Trom:  Trom-label: t3     Trom-class: Train     Current-state: S1
   AssignmentVector:
    Attribute label: cr     Attribute type: PORT_TYPE
     Port-ID: NULL
   ReactionVector:
    ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
    ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
  Triggering Transition:
   Label: R1     Source: S1     Destin: S2     Trigger Event: Near
  Enabling Reaction - Time Constraint:
   Label: TC1     Transition: R1     Constrained Event: In     Bounds: [8,16]
  Scheduling Sim     Event: In     Trom: t3     Port: NULLPORT      Time: 19
   History:
   Event Outcome: NOTYET_HANDLED

 Current Simulation Time: 7
 Handling     Sim     Event: Near     Trom: c1     Port: P3      Time: 7
   History:
   Event Outcome: NOTYET_HANDLED
  Active Trom:  Trom-label: c1     Trom-class: Controller     Current-state: C3
   AssignmentVector:
```

```
    Attribute label: inSet    Attribute type: TRAIT_TYPE
    Trait type: PSet        Trait name: Set
    Trait value:
     Port-ID: P1
   ReactionVector:
    ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
    ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
  Triggering Transition:
   Label: R2    Source: C3    Destin: C3    Trigger Event: Near

Current Simulation Time: 7
Current Simulation Time: 8
Current Simulation Time: 9
Current Simulation Time: 10
Current Simulation Time: 11
Current Simulation Time: 12
Current Simulation Time: 13
Current Simulation Time: 14
Current Simulation Time: 15
Current Simulation Time: 16
Current Simulation Time: 17
Handling    Sim    Event: In    Trom: t2    Port: NULLPORT    Time: 17
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: t2    Trom-class: Train    Current-state: S2
  AssignmentVector:
   Attribute label: cr    Attribute type: PORT_TYPE
    Port-ID: C2
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : [13,21]
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
Triggering Transition:
 Label: R2    Source: S2    Destin: S3    Trigger Event: In
Firing Reaction - Time Constraint:
 Label: TC1    Transition: R1    Constrained Event: In    Bounds: [8,16]
Enabling Reaction - Time Constraint:
 Label: TC2    Transition: R2    Constrained Event: Exit    Bounds: [0,8]
Scheduling Sim    Event: Exit    Trom: t2    Port: C1    Time: 21
 History:
```

```
  Event Outcome: NOTYET_HANDLED
 Scheduling Sim    Event: Exit    Trom: c1    Port: P2     Time: 21
  History:
  Event Outcome: NOTYET_HANDLED

Current Simulation Time: 17
Handling    Sim    Event: Out    Trom: t2    Port: NULLPORT     Time: 17
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: t2    Trom-class: Train    Current-state: S3
  AssignmentVector:
   Attribute label: cr    Attribute type: PORT_TYPE
    Port-ID: C2
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : [17,25]
 Triggering Transition:
  Label: R3    Source: S3    Destin: S4    Trigger Event: Out

Current Simulation Time: 17
Current Simulation Time: 18
Current Simulation Time: 19
Handling    Sim    Event: In    Trom: t1    Port: NULLPORT     Time: 19
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: t1    Trom-class: Train    Current-state: S2
  AssignmentVector:
   Attribute label: cr    Attribute type: PORT_TYPE
    Port-ID: C1
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : [11,19]
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
Triggering Transition:
 Label: R2    Source: S2    Destin: S3    Trigger Event: In
Firing Reaction - Time Constraint:
 Label: TC1    Transition: R1    Constrained Event: In    Bounds: [8,16]
Enabling Reaction - Time Constraint:
 Label: TC2    Transition: R2    Constrained Event: Exit    Bounds: [0,8]
Scheduling Sim    Event: Exit    Trom: t1    Port: C1     Time: 23
```

```
  History:
  Event Outcome: NOTYET_HANDLED
 Scheduling Sim    Event: Exit    Trom: c1    Port: P1    Time: 23
  History:
  Event Outcome: NOTYET_HANDLED

Current Simulation Time: 19
Handling    Sim    Event: Out    Trom: t1    Port: NULLPORT    Time: 19
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: t1    Trom-class: Train    Current-state: S3
  AssignmentVector:
   Attribute label: cr    Attribute type: PORT_TYPE
    Port-ID: C1
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : [19,27]
 Triggering Transition:
  Label: R3    Source: S3    Destin: S4    Trigger Event: Out

Current Simulation Time: 19
Handling    Sim    Event: In    Trom: t3    Port: NULLPORT    Time: 19
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: t3    Trom-class: Train    Current-state: S2
  AssignmentVector:
   Attribute label: cr    Attribute type: PORT_TYPE
    Port-ID: C1
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : [15,23]
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Triggering Transition:
  Label: R2    Source: S2    Destin: S3    Trigger Event: In
Firing Reaction - Time Constraint:
 Label: TC1    Transition: R1    Constrained Event: In    Bounds: [8,16]
Enabling Reaction - Time Constraint:
 Label: TC2    Transition: R2    Constrained Event: Exit    Bounds: [0,8]
Scheduling Sim    Event: Exit    Trom: t3    Port: C1    Time: 23
 History:
```

```
   Event Outcome: NOTYET_HANDLED
  Scheduling Sim    Event: Exit    Trom: c1    Port: P3     Time: 23
   History:
   Event Outcome: NOTYET_HANDLED

 Current Simulation Time: 19
 Handling    Sim    Event: Out    Trom: t3    Port: NULLPORT     Time: 19
   History:
   Event Outcome: NOTYET_HANDLED
  Active Trom:  Trom-label: t3    Trom-class: Train    Current-state: S3
   AssignmentVector:
    Attribute label: cr    Attribute type: PORT_TYPE
     Port-ID: C1
   ReactionVector:
    ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
    ReactionSubVector: TimeConstraint: TC2 : [19,27]
  Triggering Transition:
   Label: R3    Source: S3    Destin: S4    Trigger Event: Out

 Current Simulation Time: 19
 Current Simulation Time: 20
 Current Simulation Time: 21
 Handling    Sim    Event: Exit    Trom: t2    Port: C1     Time: 21
   History:
   Event Outcome: NOTYET_HANDLED
  Active Trom:  Trom-label: t2    Trom-class: Train    Current-state: S4
   AssignmentVector:
    Attribute label: cr    Attribute type: PORT_TYPE
     Port-ID: C2
   ReactionVector:
    ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
    ReactionSubVector: TimeConstraint: TC2 : [17,25]
 Simulation Event was not handled.
 Scheduling new Simulation Event at another port.
  Scheduling Sim    Event: Exit    Trom: t2    Port: C2     Time: 21
   History:
   Event Outcome: NOTYET_HANDLED
  Scheduling Sim    Event: Exit    Trom: c2    Port: P2     Time: 21
   History:
```

```
  Event Outcome: NOTYET_HANDLED
Handling    Sim    Event: Exit    Trom: t2    Port: C2    Time: 21
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: t2    Trom-class: Train    Current-state: S4
  AssignmentVector:
   Attribute label: cr    Attribute type: PORT_TYPE
    Port-ID: C2
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : [17,25]
 Triggering Transition:
  Label: R4    Source: S4    Destin: S1    Trigger Event: Exit
 Firing Reaction - Time Constraint:
  Label: TC2    Transition: R2    Constrained Event: Exit    Bounds: [0,8]

Current Simulation Time: 21
Current Simulation Time: 21
Handling    Sim    Event: Exit    Trom: c2    Port: P2    Time: 21
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: c2    Trom-class: Controller    Current-state: C3
  AssignmentVector:
   Attribute label: inSet    Attribute type: TRAIT_TYPE
   Trait type: PSet        Trait name: Set
   Trait value:
    Port-ID: P2
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Triggering Transition:
  Label: R5    Source: C3    Destin: C4    Trigger Event: Exit
 Enabling Reaction - Time Constraint:
  Label: TC2    Transition: R5    Constrained Event: Raise    Bounds: [0,4]
Scheduling Sim    Event: Raise    Trom: c2    Port: G1    Time: 23
 History:
 Event Outcome: NOTYET_HANDLED
Scheduling Sim    Event: Raise    Trom: g2    Port: S1    Time: 23
 History:
```

```
   Event Outcome: NOTYET_HANDLED

 Current Simulation Time: 21
 Current Simulation Time: 22
 Current Simulation Time: 23
 Handling    Sim    Event: Exit    Trom: t1    Port: C1    Time: 23
   History:
   Event Outcome: NOTYET_HANDLED
  Active Trom:  Trom-label: t1    Trom-class: Train    Current-state: S4
   AssignmentVector:
    Attribute label: cr    Attribute type: PORT_TYPE
     Port-ID: C1
   ReactionVector:
    ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
    ReactionSubVector: TimeConstraint: TC2 : [19,27]
  Triggering Transition:
   Label: R4    Source: S4    Destin: S1    Trigger Event: Exit
  Firing Reaction - Time Constraint:
   Label: TC2    Transition: R2    Constrained Event: Exit    Bounds: [0,8]

Current Simulation Time: 23
Handling    Sim    Event: Exit    Trom: c1    Port: P1    Time: 23
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: c1    Trom-class: Controller    Current-state: C3
  AssignmentVector:
   Attribute label: inSet    Attribute type: TRAIT_TYPE
   Trait type: PSet        Trait name: Set
   Trait value:
    Port-ID: P3
    Port-ID: P1
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Triggering Transition:
  Label: R4    Source: C3    Destin: C3    Trigger Event: Exit

Current Simulation Time: 23
Handling    Sim    Event: Exit    Trom: t3    Port: C1    Time: 23
```

```
   History:
   Event Outcome: NOTYET_HANDLED
  Active Trom:  Trom-label: t3    Trom-class: Train    Current-state: S4
   AssignmentVector:
    Attribute label: cr    Attribute type: PORT_TYPE
     Port-ID: C1
   ReactionVector:
    ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
    ReactionSubVector: TimeConstraint: TC2 : [19,27]
  Triggering Transition:
   Label: R4    Source: S4    Destin: S1    Trigger Event: Exit
  Firing Reaction - Time Constraint:
   Label: TC2    Transition: R2    Constrained Event: Exit    Bounds: [0,8]

 Current Simulation Time: 23
 Handling    Sim    Event: Exit    Trom: c1    Port: P3    Time: 23
   History:
   Event Outcome: NOTYET_HANDLED
  Active Trom:  Trom-label: c1    Trom-class: Controller    Current-state: C3
   AssignmentVector:
    Attribute label: inSet    Attribute type: TRAIT_TYPE
    Trait type: PSet        Trait name: Set
    Trait value:
     Port-ID: P3
   ReactionVector:
    ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
    ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
  Triggering Transition:
   Label: R5    Source: C3    Destin: C4    Trigger Event: Exit
  Enabling Reaction - Time Constraint:
   Label: TC2    Transition: R5    Constrained Event: Raise    Bounds: [0,4]
  Scheduling Sim    Event: Raise    Trom: c1    Port: G1    Time: 24
   History:
   Event Outcome: NOTYET_HANDLED
  Scheduling Sim    Event: Raise    Trom: g1    Port: S1    Time: 24
   History:
   Event Outcome: NOTYET_HANDLED

 Current Simulation Time: 23
```

```
Handling    Sim    Event: Raise    Trom: c2    Port: G1     Time: 23
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: c2    Trom-class: Controller    Current-state: C4
  AssignmentVector:
   Attribute label: inSet    Attribute type: TRAIT_TYPE
   Trait type: PSet        Trait name: Set
   Trait value: NULL-SET
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : [21,25]
 Triggering Transition:
  Label: R6    Source: C4    Destin: C1    Trigger Event: Raise
 Firing Reaction - Time Constraint:
  Label: TC2    Transition: R5    Constrained Event: Raise    Bounds: [0,4]

Current Simulation Time: 23
Handling    Sim    Event: Raise    Trom: g2    Port: S1     Time: 23
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: g2    Trom-class: Gate    Current-state: G3
  AssignmentVector: NULL-VECTOR
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Triggering Transition:
  Label: R3    Source: G3    Destin: G4    Trigger Event: Raise
 Enabling Reaction - Time Constraint:
  Label: TC2    Transition: R3    Constrained Event: Up    Bounds: [4,8]
 Scheduling Sim    Event: Up    Trom: g2    Port: NULLPORT     Time: 28
  History:
  Event Outcome: NOTYET_HANDLED

Current Simulation Time: 23
Current Simulation Time: 24
Handling    Sim    Event: Raise    Trom: c1    Port: G1     Time: 24
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: c1    Trom-class: Controller    Current-state: C4
```

```
   AssignmentVector:
    Attribute label: inSet    Attribute type: TRAIT_TYPE
    Trait type: PSet        Trait name: Set
    Trait value: NULL-SET
   ReactionVector:
    ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
    ReactionSubVector: TimeConstraint: TC2 : [23,27]
  Triggering Transition:
   Label: R6    Source: C4    Destin: C1    Trigger Event: Raise
  Firing Reaction - Time Constraint:
   Label: TC2    Transition: R5    Constrained Event: Raise    Bounds: [0,4]

Current Simulation Time: 24
Handling    Sim    Event: Raise    Trom: g1    Port: S1    Time: 24
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: g1    Trom-class: Gate    Current-state: G3
  AssignmentVector: NULL-VECTOR
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
 Triggering Transition:
  Label: R3    Source: G3    Destin: G4    Trigger Event: Raise
 Enabling Reaction - Time Constraint:
  Label: TC2    Transition: R3    Constrained Event: Up    Bounds: [4,8]
 Scheduling Sim    Event: Up    Trom: g1    Port: NULLPORT    Time: 31
  History:
  Event Outcome: NOTYET_HANDLED

Current Simulation Time: 24
Current Simulation Time: 25
Current Simulation Time: 26
Current Simulation Time: 27
Current Simulation Time: 28
Handling    Sim    Event: Up    Trom: g2    Port: NULLPORT    Time: 28
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: g2    Trom-class: Gate    Current-state: G4
  AssignmentVector: NULL-VECTOR
```

```
   ReactionVector:
    ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
    ReactionSubVector: TimeConstraint: TC2 : [27,31]
  Triggering Transition:
   Label: R4    Source: G4    Destin: G1    Trigger Event: Up
  Firing Reaction - Time Constraint:
   Label: TC2    Transition: R3    Constrained Event: Up    Bounds: [4,8]

Current Simulation Time: 28
Current Simulation Time: 29
Current Simulation Time: 30
Current Simulation Time: 31
Handling    Sim    Event: Up    Trom: g1    Port: NULLPORT    Time: 31
  History:
  Event Outcome: NOTYET_HANDLED
 Active Trom:  Trom-label: g1    Trom-class: Gate    Current-state: G4
  AssignmentVector: NULL-VECTOR
  ReactionVector:
   ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
   ReactionSubVector: TimeConstraint: TC2 : [28,32]
 Triggering Transition:
  Label: R4    Source: G4    Destin: G1    Trigger Event: Up
 Firing Reaction - Time Constraint:
  Label: TC2    Transition: R3    Constrained Event: Up    Bounds: [4,8]

Current Simulation Time: 31
Current Simulation Time: 32

     Activate Debugger? (y/n): y
     Query menu:
     0.  Return to simulation.
     1.  Display current simulation time.
     2.  Display system status.
     3.  Display subsystem status.
     4.  Display TROM status.
     5.  Display simulation event list.
     6.  Inject simulation event.
     7.  Roll-back to given time.
     8.  Activate query handler.
```

```
    9.  Activate trace analyzer.
    10. Terminate simulation.

       Select menu item: 9
    Query menu:
    0.  Return to debugger.
    1.  Display simulation events causing transition.
    2.  Display simulation events causing no transition.
    3.  Display simulation events not yet handled.
    4.  Display simulation events for given period.
    5.  Display simulation events for given Trom.
    6.  Display simulation events for given Trom & period.
    7.  Display system status at given time.
    8.  Display status for given subsystem & time.
    9.  Display status for given Trom & time.
    10. Terminate simulation.

       Select query number: 8

    Please enter subsystem label: TCG
    Please enter time-point: 15

Subsystem label:TCG
Subsystem status at time 15:
 Trom-label: t1    Trom-class: Train    State: S2
 AssignmentVector:
  Attribute label: cr    Attribute type: PORT_TYPE
   Port-ID: C1
 ReactionVector:
  ReactionSubVector: TimeConstraint: TC1 : [11,19]
  ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
Trom-label: t2    Trom-class: Train    State: S2
AssignmentVector:
 Attribute label: cr    Attribute type: PORT_TYPE
  Port-ID: C2
ReactionVector:
 ReactionSubVector: TimeConstraint: TC1 : [13,21]
 ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
Trom-label: t3    Trom-class: Train    State: S2
```

```
AssignmentVector:
 Attribute label: cr    Attribute type: PORT_TYPE
  Port-ID: C1
ReactionVector:
 ReactionSubVector: TimeConstraint: TC1 : [15,23]
 ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
Trom-label: c1    Trom-class: Controller    State: C3
AssignmentVector:
 Attribute label: inSet    Attribute type: TRAIT_TYPE
 Trait type: PSet         Trait name: Set
 Trait value:
  Port-ID: P3
  Port-ID: P1
ReactionVector:
 ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
 ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
Trom-label: c2    Trom-class: Controller    State: C3
AssignmentVector:
 Attribute label: inSet    Attribute type: TRAIT_TYPE
 Trait type: PSet         Trait name: Set
 Trait value:
  Port-ID: P2
ReactionVector:
 ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
 ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
Trom-label: g1    Trom-class: Gate    State: G3
AssignmentVector: NULL-VECTOR
ReactionVector:
 ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
 ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR
Trom-label: g2    Trom-class: Gate    State: G3
AssignmentVector: NULL-VECTOR
ReactionVector:
 ReactionSubVector: TimeConstraint: TC1 : NULL-VECTOR
 ReactionSubVector: TimeConstraint: TC2 : NULL-VECTOR

   Query menu:
   0.  Return to debugger.
   1.  Display simulation events causing transition.
```

```
 2.  Display simulation events causing no transition.
 3.  Display simulation events not yet handled.
 4.  Display simulation events for given period.
 5.  Display simulation events for given Trom.
 6.  Display simulation events for given Trom & period.
 7.  Display system status at given time.
 8.  Display status for given subsystem & time.
 9.  Display status for given Trom & time.
10.  Terminate simulation.

   Select query number: 0

Query menu:
 0.  Return to simulation.
 1.  Display current simulation time.
 2.  Display system status.
 3.  Display subsystem status.
 4.  Display TROM status.
 5.  Display simulation event list.
 6.  Inject simulation event.
 7.  Roll-back to given time.
 8.  Activate query handler.
 9.  Activate trace analyzer.
10.  Terminate simulation.

   Select menu item: 8

Query menu:
 0.  Return to debugger.
 1.  Display Trom AST.
 2.  Display transitions for given Trom.
 3.  Display transitions from current state.
 4.  Display transitions from given state.
 5.  Display transitions to given state.
 6.  Display transitions by given event.
 7.  Display time constraints for given Trom.
 8.  Display time constraints for a trigger event.
 9.  Display time constraints for a constrained event.
10.  Terminate simulation.
```

```
    Select query number: 2

   Please enter Trom label: c1

Transition Label: R1    Source: C1    Destin: C2    Trigger Event: Near
Transition Label: R2    Source: C2    Destin: C2    Trigger Event: Near
Transition Label: R2    Source: C3    Destin: C3    Trigger Event: Near
Transition Label: R3    Source: C2    Destin: C3    Trigger Event: Lower
Transition Label: R4    Source: C3    Destin: C3    Trigger Event: Exit
Transition Label: R5    Source: C3    Destin: C4    Trigger Event: Exit
Transition Label: R6    Source: C4    Destin: C1    Trigger Event: Raise

   Query menu:
   0.  Return to debugger.
   1.  Display Trom AST.
   2.  Display transitions for given Trom.
   3.  Display transitions from current state.
   4.  Display transitions from given state.
   5.  Display transitions to given state.
   6.  Display transitions by given event.
   7.  Display time constraints for given Trom.
   8.  Display time constraints for a trigger event.
   9.  Display time constraints for a constrained event.
   10. Terminate simulation.

      Select query number: 0

   Query menu:
   0.  Return to simulation.
   1.  Display current simulation time.
   2.  Display system status.
   3.  Display subsystem status.
   4.  Display TROM status.
   5.  Display simulation event list.
   6.  Inject simulation event.
   7.  Roll-back to given time.
   8.  Activate query handler.
   9.  Activate trace analyzer.
```

```
     10. Terminate simulation.

        Select menu item: 10

Simulation completed.

Exiting TROMLAB Environment.
```

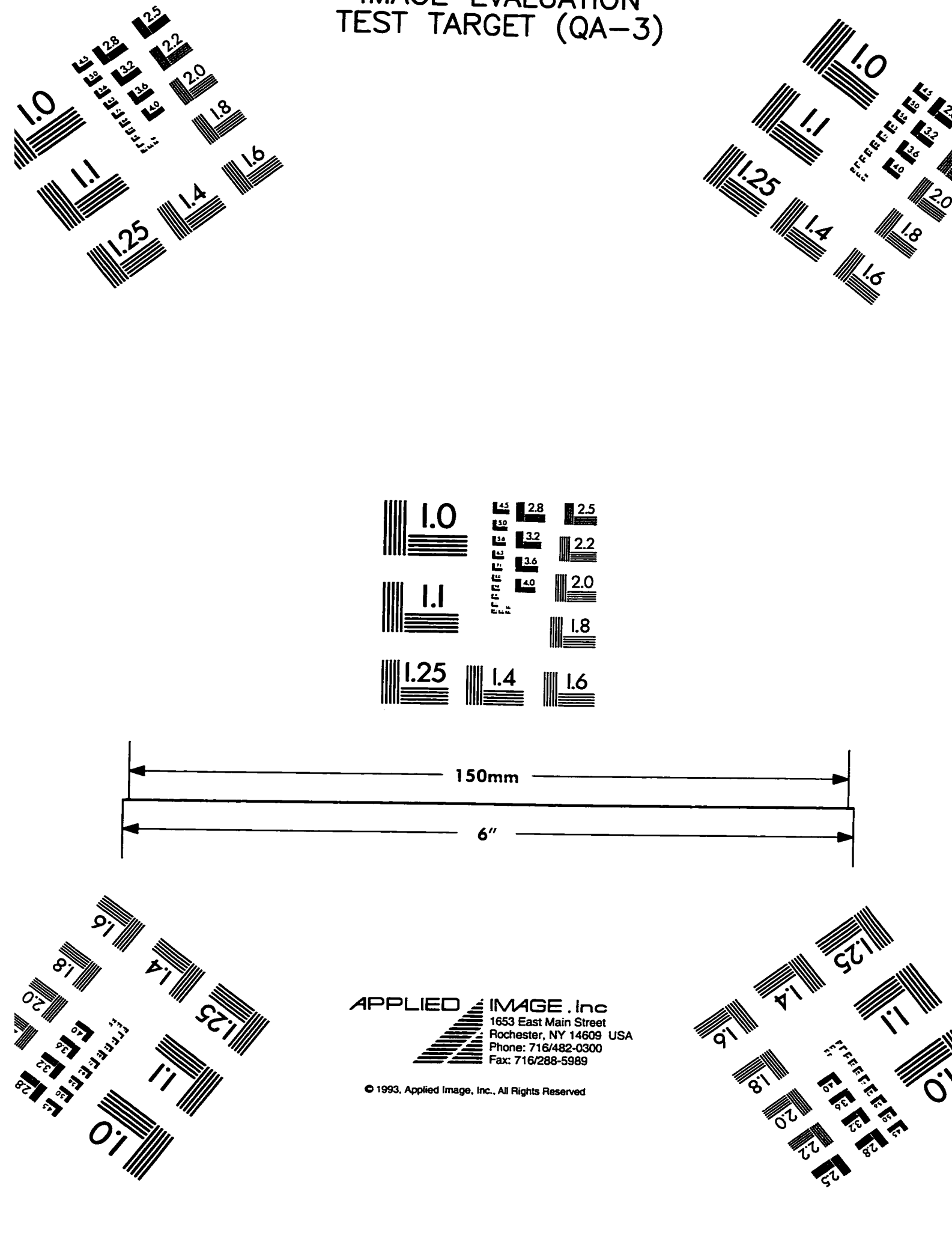


IMAGE EVALUATION
TEST TARGET (QA-3)
1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
150mm
6"
APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989
© 1993, Applied Image, Inc., All Rights Reserved