REAL–TIME REACTIVE SYSTEM DEVELOPMENT – A FORMAL APPROACH BASED ON UML AND PVS

DARMALINGUM MUTHIAYEN

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

JANUARY 2000

0-612-47715-0

Abstract

Real–Time Reactive System Development – A Formal Approach based on UML and PVS

Darmalingum Muthiayen, Ph.D.
Concordia University, 2000

The notion of real-time reactive behavior encompasses concurrency, communication through sensors and actuators, and relations between input and output over time. Real-time reactive systems are inherently complex, and often used in safety-critical contexts. Application domains include control systems for nuclear reactors, air traffic, railroad crossing, telecommunications, and medical devices. Applying formal methods in the development process is seen as a means for dealing with the complexity, and for quality assurance. One of the goals is to formally verify time-dependent safety properties in the design.

The scope of this thesis encompasses three major components. We develop a visual technique for object-oriented modeling of real-time reactive systems, based on a minimal set of extensions to UML, along with a set of well-formedness rules for the real-time models. We then present a formalization of the Real-Time UML (RTUML) notation, making use of the abstract syntax and well-formedness rules of UML metamodel, and provide formal denotational and operational semantics for RTUML. Finally, we introduce a methodology for mechanized verification of time-dependent properties in the RTUML design of real-time reactive systems, within the PVS verification environment. The formal semantics of RTUML provides a foundation for the verification methodology, and for rigorous analysis and validation techniques. The novelty of the development methodology for real-time systems lies in the mechanized verification approach superimposed on the object-oriented modeling technique.

To my parents.

Acknowledgments

My supervisor, Professor V. S. Alagar, provided continuous support, both technical and financial, throughout my studies. His shrewdness and perspicacity have been fundamental to the success of this work. Dr. Alagar has the ingenuity to extricate breakthroughs from perplexities. In addition to research coordination, Dr. Alagar provides a framework imbued with commitment, dedication, ethical standards and conviviality. His spiritual and philosophical background, coupled with utmost kindness, bring about a sound environment conducive to growth. For guiding me through this phase of my life, I am forever indebted to him. Dr. Alagar reviewed a preliminary version of this thesis and provided useful comments.

I gratefully acknowledge the fellowships provided by the Natural Sciences and Engineering Research Council (NSERC) of Canada and Le Fonds pour la Formation de Chercheurs et l'Aide a la Recherche (Fonds FCAR) of Quebec, and the External Award Holder Doctoral Scholarship awarded to me by Concordia University. I am thankful to the IEEE Computer Society for bestowing upon me the Richard E. Mervin Scholarship.

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 all, with special mention for Stan Swiercz for his commitment, dedication and efficiency. I thank the secretaries of the department for their courteous collaboration.

Thanks go to Oana Popistas for clarifying discussions on modeling using Rational Rose, which brought about improvements to the initial version of the RTUML modeling technique. Oana developed a translator for generating TROM formal specifications from RTUML models in the Rose environment. Thanks go to Francois Pompeo for clarifying discussions on axiomatic description of design specifications. Francois implemented a translator based on the verification methodology for generating axioms in the specification language of PVS from TROM formal specifications.

I am grateful to the TROMLAB research group for thought-provoking discussions. I am thankful to the referees of the journals IEEE Transactions on Software Engineering and ACM Transactions on Software Engineering and Methodology for their insightful reviews of papers I submitted for publication.

Dr. Manas Saksena was the first person to mention about UML, and to suggest working on modeling Real-Time systems in UML and on its formal semantics. Dr. Saksena always made himself available to me; not only would he answer my questions, but he would take the lead in initiating discussions, and in raising questions to follow up on an idea or discussion. I am forever grateful to him. Dr. Kasi Periyasamy has continuously expressed encouragement, and collaborated in formal methods research. Dr. Ferhat Khendek promptly intervened whenever there was a need for technical support.

I am thankful to Dr. Alagar and Dr. Periyasamy for associating me with the publication of their book, *Specification of Software Systems*, whose reading provided me with valuable formal methods background.

Conchita and Sunil Beeharry-Panray have cared to follow my progress throughout. Sandra and Sanjaye Ramdoyal, and Corinne and Paul Fieldhouse brought their warmth and friendship.

My parents, and my sisters and their families, bring significance to the accomplishment. I am thankful to Radha for accompanying me through this journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reactive systems are characterized by continuous interaction with their environment through stimulus-response behavior. In the case of real-time reactive systems, timing constraints regulate the behavior. Control systems for nuclear reactors, air traffic and railroad crossing are typical examples of safety-critical real-time reactive systems involving concurrency and synchronous communication among reactive entities through sensors and actuators. The notion of reactive behavior encompasses relations between input and output over time, complex sequencing of events and the way the events constrain computations. This thesis describes a methodology based on a synthesis of object-oriented and real-time technologies for rigorous development of real-time reactive systems.

1.1 Real-Time Reactive Systems

The distinguishing characteristic of a *reactive system* is its continuous interaction with the environment in which it resides through stimulus-response behavior. A process reacts to event occurrences corresponding to stimuli from the environment. The sequence of interactions depends on several factors, the most influential being the level of coupling between the entities in the environment. In *real-time* reactive systems, time constraints regulate the stimulus-response behavior. The notion of *reactive behavior* encompasses concurrency, communication through sensors and actuators, and relations between input and output over time. Real-time reactive systems often operate in *safety-critical* contexts, with applications in control systems for nuclear reactors, air traffic, railroad crossings, telecommunications, 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.

Factors contributing to the complexity of real-time reactive systems include size, criticality, concurrency, and the time-dependent nature of artifacts they control. Interactions in a reactive system can be complex; an entity may interact with several other entities to evoke a time-constrained behavior within a subsequent time interval, if environmental conditions hold. Such behavior exhibits *nondeterminism* in *time*, *control*, and *interaction*. Since a real-time reactive system for safety-critical applications requires off-line validation and verification, the specification language should be such that the design can be subjected to a formal analysis.

The object-oriented paradigm supports the structuring of specifications into independent and reusable components, providing a means for tackling complexity. The instantiation mechanism partitions the universe of objects into classes; inheritance and subtype relations among classes support incremental development; and aggregation relations allow building large systems by composing modules. The message passing paradigm can be exploited for describing interaction among system components. Mechanisms such as asynchronous communication, remote procedure call, and synchronous rendez-vous fit well in the message passing paradigm and provide an appropriate basis for dealing with concurrency.

Real-time reactive systems can be viewed as real-time concurrent systems comprising of communicating processes. For instance, in the *railroad crossing* problem [HL94], a train informs the controller monitoring the crossing that it is approaching the gate. Upon receiving the message, the controller instructs the gate to close within a prescribed time bound, and the gate lowers its arm within a prescribed time window. The problem incorporates real-time constraints between environmental objects as well as system objects. More complex timing constraints, such as "*during the period that a train is crossing a gate, the controller should be monitoring the gate, and the gate must remain closed*", involve several objects and cannot be stated only in terms of constraints on the computations of individual objects. The superposition of real-time constraints on the message passing mechanism in the object-oriented paradigm is suitable for handling modularity, compositionality and concurrency in the development of real-time reactive systems. Since a complete and correct set of environmental and system requirements is not available a priori, the development process should effectively handle evolving requirements. Object-orientation adequately supports adaptability.

- Polymorphic modules allow component substitution without affecting coupled components.
- Inheritance can factor out common features; hence modifications would often be in one place.
- With encapsulation, changes tend to be somewhat localized.

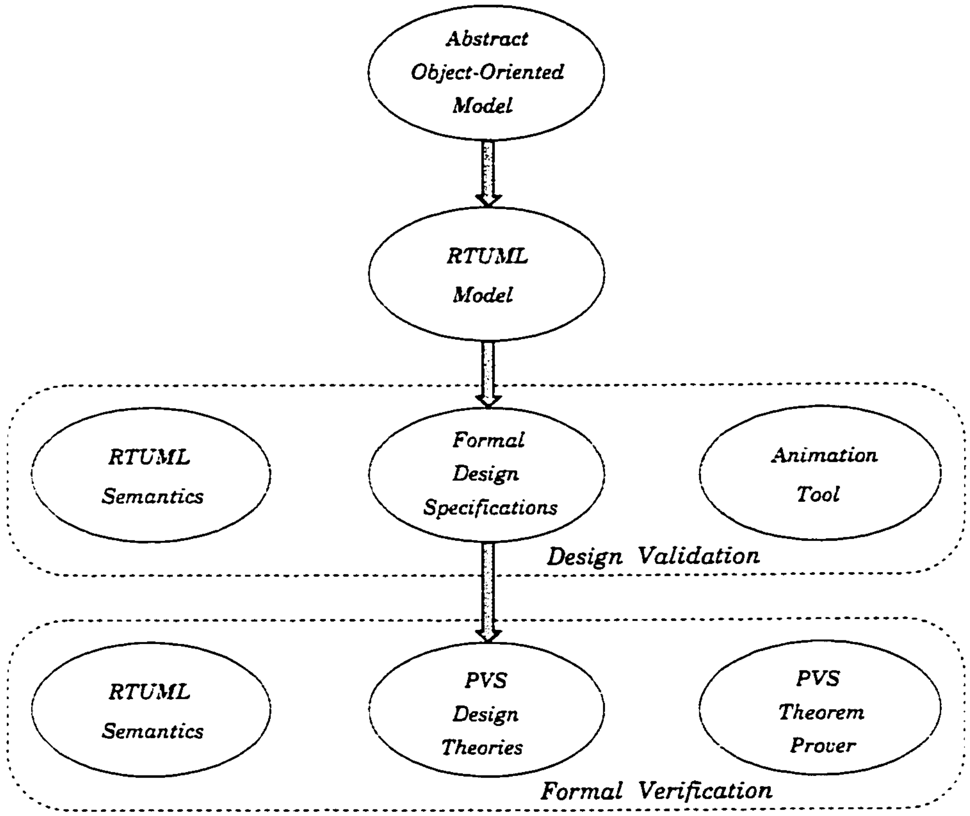


Figure 1: Validation and Verification Goals for RTUML.

1.2 Research Goals

This thesis proposes a methodology based on a synthesis of object-oriented and real-time technologies for rigorous development of dependable real-time reactive systems. The methodology purports to visual modeling in the industrial standard graphical notation UML (Unified Modeling Language) [OMG99], and mechanized verification of time-dependent properties in object-oriented design specifications using the verification system PVS (Prototype Verification System) [ORS92]. The work integrates UML and PVS by providing denotational and operational semantics for the UML-based modeling technique for real-time reactive systems in the specification language of PVS.

The formalization of UML is undertaken to fulfill the need for a sound foundation for requirements modeling and rigorous design analysis in the context of safety-critical systems. The motivation for this work comes from two fronts:

- the wide acceptance of UML in industry, as a unified notation applicable to the development of objects in a broad spectrum of domains, and
- the use of PVS for design analysis in industrial scale applications, as reported in NASA guidebooks [NAS95] [NAS97].

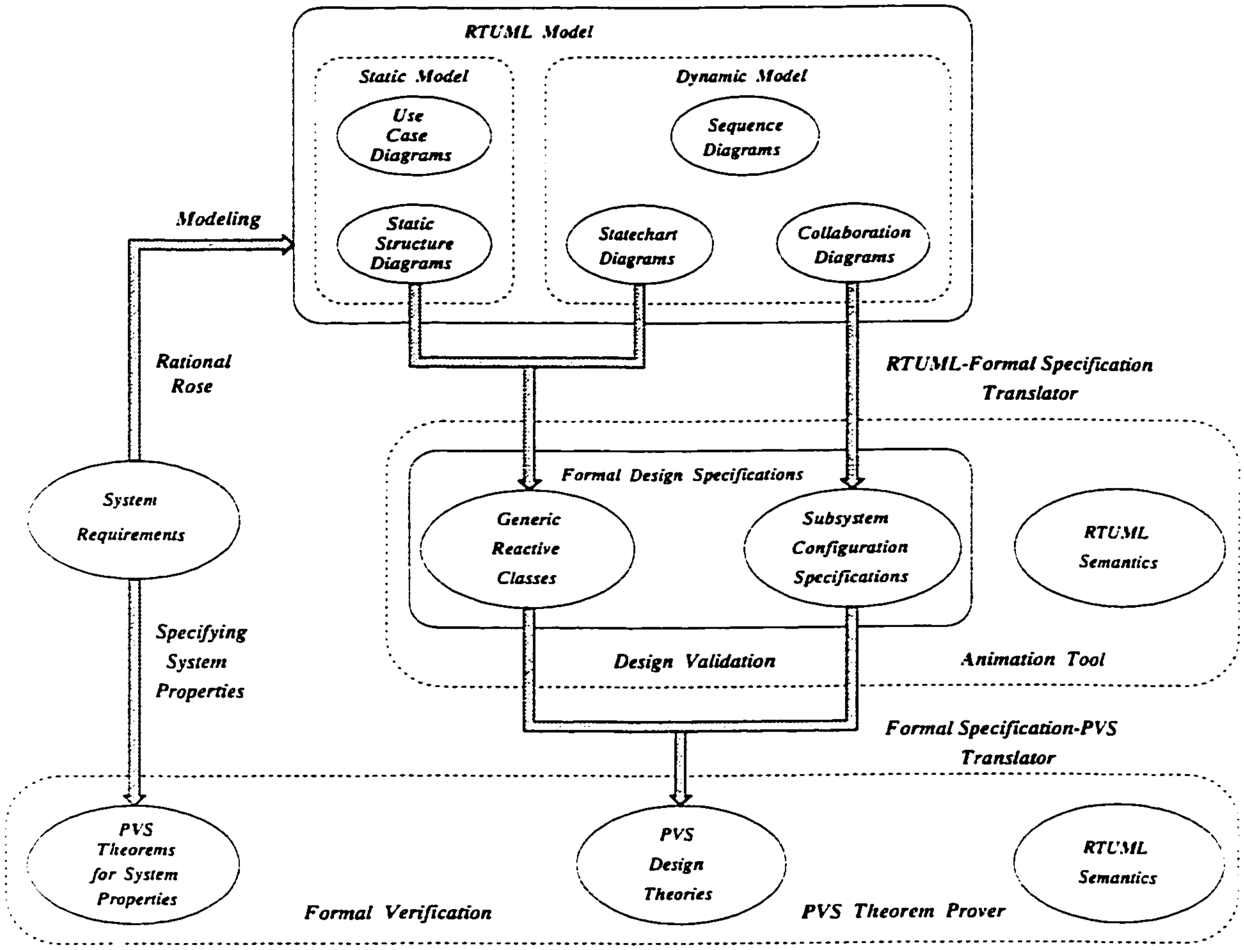


Figure 2: Artifacts for Validation and Verification.

Our goals are

- to provide a UML-based visual modeling technique for real-time reactive systems, within a formal framework supporting the automated generation of formal specifications for simulation, reasoning, and validation purposes,
- a formal semantics for the visual modeling technique so as to provide a foundation for rigorous analysis methods, and
- a mechanized verification methodology for proving time-dependent properties in the design specifications generated from the UML-based models.

Figure 1 illustrates the main stages in the underlying development process. From an abstract object-oriented model of the system under development, we produce a visual model in *RTUML (Real-Time UML)*, the proposed UML hybrid for real-time systems, using Rational Rose. A translator extracts formal object-oriented design specifications from the visual model. These specifications can be typechecked and simulated within the animation tool of TROMLAB development environment for validation purposes. Subsequently, a second translator generates an axiomatic description of the system's behavior in PVS specification language. These axioms are used for proving time-dependent properties in the design, using PVS theorem prover. Figure 2 provides a more detailed illustration of these design analysis stages.

1.2.1 Extending UML for Real-Time Systems

The contributions of this research are

- a minimal set of extensions to UML notation to engraft an abstract model for real-time reactive systems,
- a translation mechanism from a UML model of a reactive system onto formal specifications.

Applying UML extension mechanisms, we introduce stereotypes on classes and associations to capture an abstract object-oriented model for real-time reactive system design. We adapt UML statechart diagram so that the *guard* and *action* concepts reflect the logical assertions and timing constraints associated with transitions. The resulting UML hybrid notation, RTUML, incorporates class and statechart diagrams for object models, and collaboration and sequence diagrams for subsystem models. In this exercise, we have avoided to introduce new symbols in the notation, making the graphic model concordant with the existing diagrams. For rigorous analysis of system designs, it is imperative to provide a sound foundation for the description of the models. We provide well-formedness rules in OCL for the abstract model, based on the UML metamodel. Since UML abstract syntax and well-formedness rules are defined in OCL, the well-formedness rules bring forth a formal framework for the specification technique. In defining a translation mechanism, we map the visual reactive object model onto a textual description of generic reactive classes and subsystems, using a formal definition of the model as the basis. Deriving formal specifications from the visual model of a reactive system supports simulation of the design to validate requirements, and extraction of axioms to conduct formal verification of desired system properties.

1.2.2 Formal Semantics for Real-Time UML

We provide formal semantics for the Real-Time UML notation, and embed the notation in PVS specification language. The formalization of UML is undertaken to fulfill the need for a sound foundation for requirements modeling and rigorous design analysis in the context of safety-critical systems. Formalization of the semantics integrates the graphical notation of object modeling techniques with formal methods. One of the goals of providing formal semantics is to serve as a specification environment for design analysis of objects, classes and subsystem models. In order that the modeling and analysis can be done in a rigorous manner, the initial steps involve formalizing the modeling language, providing formal semantics, and instituting mechanisms for checking completeness and consistency.

Our approach to defining formal semantics for RTUML notation has two aspects, denotational semantics and operational semantics. In the process of defining denotational semantics, we first formalize UML metamodel, including its abstract syntax and well-formedness rules in PVS specification language, and incorporate the set of extensions in terms of additional elements in the sets of model element stereotypes. We then describe a set of restrictions on the core UML notation, in the form of well-formedness rules for RTUML, and specify these in OCL on the abstract syntax of the core UML notation. Subsequently, we translate the

RTUML well-formedness rules in the specification language of PVS. This provides a complete formalization of the syntax of RTUML in PVS.

The next step is to define the semantic domain in terms of concepts from the abstract reactive object model, such as generic reactive class, configuration, scenario, and reactive system model. The correctness of a semantic domain concept is defined in terms of a set of axioms for each concept. We perform this exercise in a set-theoretic approach to the domain of types. These definitions are then formalized in the specification language of PVS. Having formalized RTUML syntax and the semantic domain, the last step is to define a mapping from the syntactic constructs to the semantic domain concepts. We ensure completeness of the semantic definition by providing a mapping for each model element used in the RTUML notation. We finally formalize the semantic mapping in terms of PVS functions for each model element, and axioms characterizing the mapping.

Having defined the denotational semantics of RTUML, we provide an operational semantics based on an axiomatization of the abstract reactive object model. We first specify each axiom in OCL (Object Constraint Language) [OMG99], to integrate it within the UML framework, and then provide a PVS specification of the axioms to fit into the overall formal semantics framework. The operational semantics is subsequently used in providing a foundation for the mechanized verification methodology described below.

1.2.3 Mechanized Verification of Time-Dependent Properties

The methodology derives axioms involving linear inequalities over absolute times for transitions from the design specifications of a reactive system, and proves that a time-dependent property is a logical consequence of the axioms. We derive axicms specifying (i) ordering relations on transitions, (ii) timing constraints on reactions to a transition, (ii) synchronization of message exchanges, and (iv) other axioms specific to the requirements of the system. A desired property corresponds to an invariance assertion on the design specifications. We formulate a theorem specifying the assertion as an expression involving linear inequalities over absolute times for event occurrences. Within the PVS specification and verification environment, we define a theory including the axioms and theorem, and establish the property by constructing a proof for the theorem. An automated derivation process yields the first three kinds of axioms; the proof construction process uses PVS logic and theorem prover.

We outline the logical foundations of the methodology with respect to the behavioral semantics of the underlying object-oriented formalism. Applying the semantics to the formal model of a reactive system, we derive a set of axioms involving state predicates and time intervals between state transitions. We use Shankar's *since* operator [Sha93] to specify durations over state predicates. Applying the *duality* between event occurrences and state transitions, we establish an equivalence between this set of axioms and the set of axioms involving absolute times for event occurrences. Similarly, the invariance assertion describing the desired property can be formalized as an expression involving the *since* operator and state predicates. The theorem to be proved corresponds to a similar transformation of this expression into a formula involving linear inequalities over absolute times for event occurrences.

1.3 Thesis Outline

Chapter 2 describes an overall development methodology for real-time reactive systems. It includes outlines of an abstract real-time reactive object model, a formalism based on the abstract model, a software development environment built upon the formalism, and a process model where the software engineering stages are supported by the development environment. The chapter also includes sketches of the UML modeling language and the PVS verification system. Chapter 3 provides a survey of work related to object-oriented modeling of real-time reactive systems, formal semantics for object modeling techniques, and formal verification of real-time systems. Chapter 4 describes RTUML, the UML-based object-oriented specification technique for real-time reactive systems. It outlines the extensions brought to UML, along with well-formedness rules for the hybrid notation, for capturing real-time behavior. It includes a case study based on a distributed navigation controller to illustrate the visual modeling technique and how formal specifications can be derived from the visual models. Chapter 5 describes denotational semantics for the UML-based modeling technique for real-time reactive systems, and operational semantics for the abstract reactive object model in the specification language of PVS. This exercise includes defining the semantic domain for real-time systems, and a semantic mapping from the model elements in the notation to concepts in the semantic domain. It includes an OCL specification of the operational semantics for the abstract reactive object model. Chapter 6 introduces a methodology for mechanized verification of time-dependent properties in the UML-based design of real-time systems. It includes a description of how an axiomatic description of a real-time system in PVS specification language can be derived from formal design specifications of a real-time system. Chapter 7 concludes the thesis with an outline of the research contributions, and future work.

Chapter 2

Background

The TROMLAB development environment for real-time reactive systems integrates formal methods in every stage of the development process. It supports a process model for iterative development, and provides facilities for modular design of generic reactive classes, modular composition of objects to build subsystems, and analysis of system behavior, combining simulation and verification. The Unified Modeling Language is being widely accepted as an object-oriented design notation for industrial applications. However, several aspects of the notation need to be made precise for it to be accepted as a language supporting rigorous analysis. The Prototype Verification System is being used for verification of complex software systems, especially in the aeronautics industry.

2.1 Introduction

This chapter is organized as follows. Section 2.2 describes a development methodology for real-time reactive systems, that is centered around the TROMLAB software development environment. The section includes a description of an abstract real-time reactive object model, and gives an informal description of the TROM formalism, including its syntax. It briefly describes the computational model on which the simulator is built, and on which the proposed verification methodology is based. Finally, it outlines the stages in the underlying process model. Section 2.3 outlines the characteristics of the Unified Modeling Language, and sketches its metamodel, abstract syntax, and well-formedness rules. Section 2.4 gives a brief introduction to the Prototype Verification System, its specification language, and its reasoning system.

2.2 Development Methodology

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 benefits of this methodology include the execution of formal specifications prior to implementation, rigorous analysis of design specifications, and verification of time-dependent properties in the design. 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.

A significant aspect of the proposed development methodology for real-time reactive systems is the integration of formal methods with UML notation. 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, imprecisions, 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 abstract reactive system model [Ach95] has three tiers: (i) mathematical abstractions of data models used in specifying a reactive object, (ii) reactive objects with time-constrained stimulus-response behavior, and (iii) object collaborations. The three tiers independently specify abstract data types, generic reactive classes, and system configurations, respectively. Abstractly, a reactive object is a hierarchical finite state machine augmented with ports, attributes, logical assertions, and time constraints. Reactive class specifications include abstract data types specified as LSL (Larch Shared Language) [GH93] traits. Ensuring consistency and completeness of data type specifications involves constructing proofs using Larch Prover. Hoare-style specifications for state transitions may involve assertions on operations from the abstract data types. The specification style supports nondeterminism, allowing refinements and reuse. A subsystem specifies instantiations of generic reactive classes, links to configure interaction channels among the objects, and compositions of subsystems.

The formalism is sufficiently expressive for modeling reactive systems. The benefits derived from the object-oriented technique 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.

2.2.1 Abstract Real-Time Reactive Model

Identifying the requirements of a reactive system involves stating properties that are expected of the system. Since properties hold over a period of time, it is more appropriate to associate them with time intervals rather than with time points. For instance, the property "*as long as the pressure is high, the safety valve remains open*", implicitly projects an interval notion of time. A property holding over a time interval can be refined to a more detailed level that introduces subintervals. However, an occurrence of an event takes an infinitely small amount of time. In our model, an event f occurring at time t may trigger another event e to occur at some point during a finite interval [a,b], $b \geq a \geq 0$, relative to time t. That is, although actions may take an interval of time, and properties may hold over an interval of time, process reactions occur at points in a discrete time domain.

Unit of Modeling

A reactive object can be conceived through a model capturing two distinctive aspects, structure and behavior. We encapsulate the properties of a reactive object in an augmented hierarchical finite state machine. A reactive object is assumed to have a single thread of control. The communication mechanism is based on *synchronous message passing*. A message involves an *event* occurrence at a *port* of the reactive object, signifying a transmission; the recipient object receives the message instantaneously. The processes involved in a communication are tightly coupled in conformance with the synchrony hypothesis of Berry and Gonthier [BG92a]. A *port* abstracts an access point for bidirectional communication between reactive objects. The *port type* dictates the set of messages allowed at the port. Instances of a generic reactive class conform to the same functional and temporal behavior; their structure differ only in the number of ports for each port type.

A reactive object consists of port types, events, states, attributes, LSL traits, an attribute function, transition specifications, and time constraints. A state can be simple, or complex with substates. The attribute function defines the association of attributes to states; for a computation associated with a transition entering a state, only the attributes associated with that state are modifiable. A transition specification describes the computational step associated with the occurrence of an event. The values of the attributes disambiguate non-deterministic transitions. A time constraint associates a reaction with a transition: the reaction corresponds to firing an output or an internal event within a time interval subsequent to the transition. An occurrence of the transition causes the constrained event to be enabled; the enabled reaction is disabled if the object enters one of the *disabling* states associated with the time constraint.

An input event results from an incoming interaction defined by an external stimulus, the current state of the object, and the port constrained by the *port-condition* on the transition. An event occurrence causes a computation, updating the current state and the attributes specified by the attribute function. The port-condition constrains the ports at which an interaction can occur based on the values of the attributes. A state update may result in the disabling of an outstanding reaction. Based on the status of a global clock, an outstanding reaction may be fired in the form of a transition. An output event fired as a result of a reaction corresponds to a response through a port specified by the port-condition of the transition specification.

This abstract model of a reactive object incorporates four kinds of nondeterminism: (i) *Control nondeterminism*: at any state, there may be a number of valid choices concerning the transition to be fired; (ii) *Interaction nondeterminism*: at any time, there may be a number of choices concerning the port at which the event associated with the transition to be fired can occur, as specified by the port-condition. A reactive object can implicitly exhibit nondeterminism in selecting the entity in its environment with which to interact; (iii) *Timing nondeterminism*: the time constraints associated with an object specify minimum and maximum time delays between trigger and response, allowing the object to choose the appropriate delay to exhibit; (iv) *Computation nondeterminism*: the computation associated with each transition, as specified by the postcondition, can be abstract and include nondeterministic constructs.

The port-condition together with the temporal ordering of events asserted by the state machine provides a means for specifying patterns of interaction between objects in the system. The model allows the specification of several typical real-time features such as minimal and maximal delays, exact occurrences, and periodicity of event occurrences, in combination with temporal relations on stimulus and response. The model also provides encapsulation of timing constraints by precluding an input event from being a constrained event, precluding a reactive object from enforcing any timing constraint on the occurrence of input events, since these are under the control of the environment.

2.2.2 TROM – Timed Reactive Object Model

Informally, an object defined in TROM [Ach95] consists of the following elements:

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

```
Class < identifier > [< port types >]
    Events:
    States:
    Attributes:
    Traits:
    Attribute-Function:
    Transition-Specifications:
    Time-Constraints:
end
```

Figure 3: Template for Class Specification.

```
Subsystem < identifier >
    Include:
    Instantiate:
    Configure:
end
```

Figure 4: Template for System Configuration Specification.

A formalization of the abstract real-time reactive model is necessary in order to provide a formal semantics to support rigorous design analysis, and reasoning about time-dependent properties such as safety and liveness. A formal definition of the components of the abstract real-time reactive object model is available in [Ach95]. The grammar of the specification language is a direct derivative of the formal definition of a reactive object. 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. Figure 3 shows the syntax for formal description of a generic reactive class. The keyword Class introduces a reactive class with its identifier and its associated port types with parametric cardinality. The sections labeled with the keywords Events, States, Attributes, Traits, Attribute-Function, Transition-Specifications, and Time-Constraints capture the structure and behavior of instances of the reactive class.

A *system configuration specification* defines a subsystem by creating instances of generic reactive classes, and configuring the communication topology among these objects and others from imported subsystems. Figure 4 shows the syntax for describing a subsystem. The template includes the keyword Subsystem introducing the identifier for the subsystem, and sections labeled with the keywords Include, Instantiate, and Configure. The Include clause is for importing other subsystems. The Instantiate clause defines reactive objects by parametric substitution to cardinality of ports for each port type, and initializing attributes in the initial state of the object. The Configure clause defines a configuration by linking ports of communicating objects specified in the Instantiate clause and in subsystems imported through the Include clause.

The composition operator $\leftrightarrow$ sets up communication links between compatible ports of interacting objects. Two ports are compatible if the set of input (output) messages at one port and the set of output (input) messages at the other port are equal. The compatibility relationship on ports is symmetric but not transitive. A link $A_i.@c_j \leftrightarrow B_k.@p_l$ in the Configure clause connects the port c_j of object A_i and the port p_l of object B_k. The configuration mechanism allows one-to-one, one-to-many, and many-to-many relationships between communicating objects, and effectively determines the set of all message sequences in a collaboration.

TROM Computation

The status of a TROM object captures the state in which the TROM object 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 object 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 object 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 object. The current state of a TROM object, 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 object is thus encapsulated, and cannot be modified in any other way. 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.

A *computational step* [Ach95] of a TROM object is an atomic step which takes the TROM object from one status to its succeeding status as defined by the transition specifications. Every computational step of a TROM object is associated with a transition in the TROM object; and every transition is associated with either an interaction signal, an internal signal, or a *silent* signal. A computational step occurs when the TROM object 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 object 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 object 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 object is a sequence of computational steps.

Each computational step is associated with a transition in the state machine of the TROM object. 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 object, 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 object 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 object is described by the infinite sequences of computational steps it can undergo. The computation of a TROM object 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.

2.2.3 TROMLAB Development Environment

TROMLAB [AAM96] is a prototype object-oriented software development environment for real-time reactive systems. The development environment includes a front-end linked to Rational Rose for visual modeling, a simulator, a validation assistant, a verification assistant, a graphical user interface, and a browser for navigating through libraries of reusable components. 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 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 time-dependent properties of an evolving design. Figure 5 shows the architecture of the development environment. The 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.

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. 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 supports the detection of flaws in a system design. Such a model introduces *predictability* for properties that have to be maintained in the future. Sufficient information is required in the validation assistant to analyze and deduce reasons for a specific behavior. The computational *history* of event

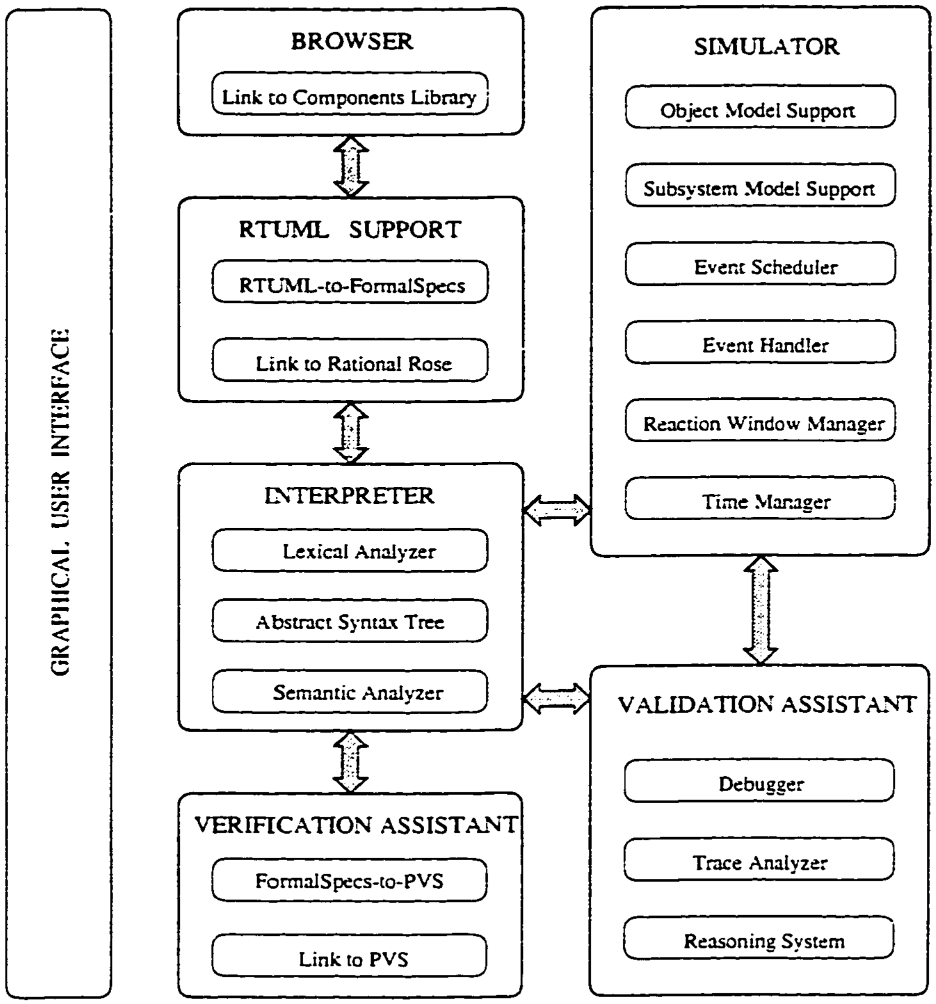


Figure 5: TROMLAB Development Environment.

traces allows the user to roll the simulation clock backward to detect and fix flaws 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*. Both validation and verification facilities are integrated in one toolset, for analyzing the behavior of the system under development during design evolution. The development environment supports modular design of reactive classes, modular composition of objects to build subsystems, and analysis capabilities which combine simulation and verification.

Animation Tool

We have developed a validation tool [Mut96, AMA96a] supporting simulation of reactive models and formal reasoning. The specification environment includes a grammar supporting the formal description of TROM classes and subsystem configurations. 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 incorporating an interpreter. The interpreter performs lexical and semantic analysis on the class definitions, and on the specification of system configurations. While parsing the 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 verification assistant to generate axioms for classes and subsystems, and by the simulator to access the static components of a TROM object. The semantic analysis allows a rigorous static inspection of the data types involved in the TROM classes and subsystems.

Debugging facilities include freezing the simulation and activating the validation tool. When the 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.

2.2.4 Process Model

The development methodology fits the process model shown in Figure 6, whose merits include iterative development, incremental design, and formalism application. The process model was introduced in [Ach95]; it has been adapted to incorporate mechanized verification. The construction of a visual model of a reactive system abstracts from functional and timing requirements, desired properties, and environmental constraints. From the visual model, a translation mechanism yields a formal model of the reactive system. The other stages involve design validation, and formal verification of the design specifications.

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 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.

Validation of the design against requirements relies on *simulation* of system behavior using the formal model. Simulation uses the formal model to generate observable behaviors that can be directly related to requirements for analysis. Consequences of refinements, changes to event occurrences, and time constraints can be analyzed before changes to the design are brought about. Flaws resulting from incorrect functionalities and inconsistent timing constraints call for a redefinition of the formal model of the reactive unit.

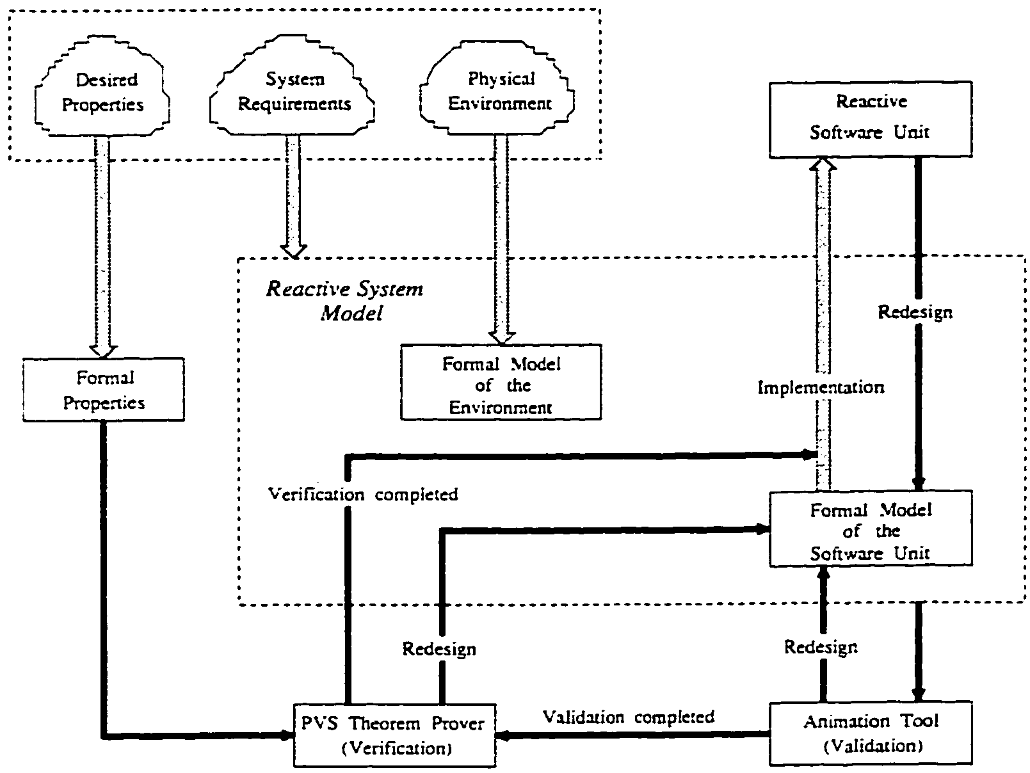


Figure 6: Process Model for Real-Time System Development.

This iterative process of redesign and simulation proceeds until only acceptable behaviors are apparent in the formal model. Facilities for simulation, debugging, analysis, and reasoning about real-time reactive systems specified according to this formalism are available in a development environment [AAM96].

Verification allows ascertaining the satisfaction of certain properties in a design to a high level of confidence. When building a system for use in safety-critical contexts, rigorous analysis of the system becomes imperative. Verification starts after concluding the validation cycle. From the formal model, we derive axioms describing the dynamic aspects of the system. The desired properties of a reactive system are usually not expressible in terms of the behavior of the unit alone; instead, they are statements about interaction between the unit and its environment. To guarantee an acceptable behavior of the unit, environmental properties on which the unit can rely have to be provided. The desired properties become theorems to be proved from the axioms specifying the dynamic model of the system. When a certain property cannot be proved, we redefine system requirements, and redesign the system to obtain a revised formal model. After completing the verification process, we proceed to develop an implementation of the reactive unit from the formal model.

2.3 UML – Unified Modeling Language

Several notations have been proposed for modeling software systems in an object-oriented style, including OMT, Fusion, and the Use Case approach. Recently, the Object Management Group undertook to unify the notations with a view to defining an industrial standard. This led to the formulation of UML as a set of modeling diagrams. This section outlines the following components of the UML notation: *static structure diagrams*, *use case diagrams*, *sequence diagrams*, *collaboration diagrams*, and *statechart diagrams*.

Static Structure Diagrams

Static structure diagrams include class diagrams and object diagrams, describing the attributes, and operations of classes and objects. Parameterized classes (templates) can be described with unbound formal parameters to describe a family of classes. The externally visible operations of a class can be specified through class interfaces. The notation introduces the classifier concept to group the following model elements: *class*, *datatype*, and *interface*. A class can be specialized into a *type* to characterize a changeable role of an object, or into an *implementation class* to define the data structure and procedures of an object. The relationship *realizes* models the implementation of a type in a programming language by an implementation class.

Static structure diagrams show relationships between classifiers, such as associations and generalizations (specializations). Types of associations include *aggregation* and *composition aggregation*. Associations can be binary, ternary, or higher-order. An association can be qualified using *qualifiers* representing attributes of the association. Multiplicity for roles within associations can be specified. An instance of an association is described as a *link*. *Composite objects* can be described, implying a composition aggregation. A generalization describes a subtyping relationship; inheritance is defined using subtypes and supertypes.

Use Case Diagrams

A use case model describes how external objects use the functionality of a system to perform a specific task. The elements of a use case model include *actors*, *use cases*, communication associations between use cases and actors, and generalization relationships among use cases. A *use case* corresponds to a unit of functionality provided by a system. An *actor* represents the role of an external object interacting directly with the system, within the context of a use case. The *communicates* relationship between an actor and a use case signifies the participation of the actor in the use case. The *extends* relationship between use cases indicates the possible inclusion of the behavior specified by a use case in an *extender* use case. The *uses* relationship between use cases indicates the inclusion of the behavior specified by a use case in a *user* use case.

A use case characterizes the set of all messages exchanged between a system and actors, as well as actions performed by the system, in the context of a task. It is an abstraction of possible scenarios for a function of the system. A scenario models an ordered sequence of messages among objects collaborating for system behavior. An instance of a use case describes the specific sequence of messages corresponding to a scenario. Different scenarios of a given use case correspond to permutations of object interactions [Dou98b].

Sequence Diagrams

A sequence diagram shows a time sequence of messages exchanged between collaborating objects as part of an interaction. The objects participating in the interaction are shown by their *lifeline*. Relationships between the objects, such as associations, are not shown. A sequence diagram can be either *generic* showing all the possible sequences of message exchange between the objects, or an *instance* of interaction showing one possible sequence of messages. A sequence diagram shows the existence and duration of the object in a role within a collaboration. Conditionality is represented by the splitting of an object lifeline into one or more concurrent lifelines. Although the time axis usually shows the sequence only, it can be used as a metric. An *activation* captures the focus of control, and indicates the performance of an action by an object. It also shows the time period for performing the action.

A *message* can correspond to a procedure call, an asynchronous message, the sending of a message from one object to another without yielding control or the yielding of a thread of control in a concurrent system, branching with guard conditions, or iteration indicating multiple occurrence of a message. The *transition time* of a message can be indicated using the sending and receiving times. These times are specified using identifiers, which can be used in expressing timing constraints. Messages can be atomic or non-atomic.

Collaboration Diagrams

A *collaboration* is a static construct representing objects involved in an operation or a use case, and relationships among the objects, such as associations, that are meaningful to the purpose. The collaboration thus describes the context in which the behavior occurs for the given operation or use case, that is the context in which interactions can occur. A collaboration can be generic, such that the participants in the collaboration can be given as parameters. A parameterized collaboration can capture a design pattern, in which case the collaboration is not attached to an operation or use case. A collaboration can be refined to obtain another one with finer granularity. An *interaction* is obtained by adding a sequence of messages to a collaboration. Different interactions can be obtained from the same collaboration, by including different sets of messages. An interaction is a behavioral specification that uses a message sequence to show how the objects in a collaboration accomplish a purpose, such as an operation or a use case.

A *collaboration diagram* represents a collaboration showing the relationships among the object roles, and an interaction among the objects. The sequence of messages exchanged among the objects and the concurrent threads are determined by the sequence numbers of the messages. Objects may be designated as *new*, *destroyed*, or *transient*, to show changes in their life state. The roles in a collaboration correspond to class references and association references, corresponding to the type of objects that can play the role. Each role is bound to an actual class supporting the operations required of the role. When the collaboration is used, each role is bound to an actual object. *Multiobjects* corresponding to sets of associated objects can be included in a collaboration. Objects can be *active*, owning a thread of control, or passive, holding data but not initiating control. The creation and destruction of objects during an interaction can be marked with the *markers*: *new* and *destroyed*.

Statechart Diagrams

A *statechart diagram* is an extended finite state machine showing all possible sequences of states through which an object or an interaction can go through in response to stimuli that it receives. It captures the responses and actions of the object or interaction. A state machine is attached to a class or a method. A *state* captures a condition during the life of an object or interaction, during which it is performing an action, waiting for an event, or satisfying a condition. Such an action is atomic and non-interruptible; however, a state may correspond to an activity, in which case the state corresponds to a nested state machine. A state can be decomposed using *and* and *or* relationships, into concurrent substates, and mutually exclusive disjoint substates respectively. An *event* corresponds to an occurrence that can trigger a state transition. Events may be of the following types: *ChangeEvent*, *SignalEvent*, *CallEvent*, and *TimeEvent*. The conditions associated with a *ChangeEvent* and a *TimeEvent* can be specified using the keywords *when* and *after* respectively.

A *transition* describes a relationship between two states such that an object in the first state will enter the second state and perform a certain action if a given condition is satisfied. Events may be parameterized, and are processed one at a time. When an event triggers more than one transition, the choice of transition may be nondeterministic. A *complex transition* may have multiple source states and target states, representing synchronization or splitting of control into concurrent threads. Such a transition is enabled only when all the source states are occupied. All the target states are occupied after the transition fires.

A transition to a complex state signifies a transition to the initial state of the complex state, or to the initial state of each of its concurrent substates. A message is sent by an action in an object to a target set of objects. An internal transition is one in which the transition remains within a single state. It corresponds to the occurrence of an event that does not cause a change of state. A self-transition involves the execution of the *exit* and *entry* actions on the state, and the initial state to be entered. An internal transition does not involve the exit and entry actions, and does not cause a state change.

2.3.1 Critical Review of UML Notation

For a rigorous analysis of design specifications, it is imperative that relationships between components of the notation are brought out. These relationships represent constraints on the model, and should be analyzed to ensure consistency across UML diagrams capturing different aspects of the design. We analyze UML notation to gather insight into how inconsistencies can be avoided. We then compare UML notation with the TROM formalism [Ach95], and analyze the commonalities between the two notations.

Relationships among Components of UML Notation

UML uses an assortment of diagrams to capture different aspects of system design; however, certain model elements occur in more than one diagram. Such relationships among components of the notation are not brought out by the semi-formal semantics given in the UML Semantics document [OMG99]. Consequently, it is hard to establish consistency among the modeled components and to ensure the presence of requirements (properties) across UML diagrams capturing a system design. Inconsistencies among different components of a design may remain undetected. To remedy this situation, we identify and formalize the relationships among the components of the notation. The formal description of these relationships is included in the formalization.

Static structure diagrams capture the static features of objects, including their attributes and operations, and relationships such as associations and generalizations among classes and objects. The static structure of an overall system design can be specified in terms of these diagrams. On the other hand, collaboration diagrams depict collaborations among sets of objects cooperating to perform specific tasks, as well as interactions capturing sequences of messages exchanged between the objects in a collaboration. It can be observed that collaborations showing associations among objects can be projected from the static structure diagrams describing the overall system.

Sequence diagrams include time sequences of messages exchanged between objects to capture interactions. Collaboration diagrams include messages exchanged between objects during interactions. Messages in collaboration diagrams are tagged with a sequence number, such that the sequence of occurrence of messages in an interaction can be derived. The sequence of messages derived from a collaboration diagram corresponds to the sequence of messages contained explicitly in the sequence diagram capturing the same interaction. Accordingly, the set of objects included in a collaboration diagram matches the set of objects included in the corresponding sequence diagram.

Statechart diagrams represent sequences of states in which an object or interaction goes through during its lifetime. Transitions from one state to another result from the occurrence of certain events. When more than one object is involved in an event occurrence, the event corresponds to a message exchanged between the objects. Consequently, this event can be mapped onto a message in the corresponding interaction diagrams (sequence diagram and collaboration diagram). Moreover, timing constraints on the occurrence of events, and on responses to such events, given in statechart diagrams must match those given in sequence diagrams.

Comparison of UML Notation with TROM Notation

We analyze the static structure of TROM models, as well as its dynamic aspects, to compare the design notation with corresponding UML constructs. Constructs corresponding to the static features of a TROM class, namely lists of port types, ports, events, states, attributes, attribute functions, and LSL traits, can be captured in UML static structure diagrams and statechart diagrams extended with timing constraints on transitions. A composition aggregation association can be used to describe the incorporation of instances of an LSL trait in a TROM class.

Transition specifications, defining the dynamic behavior of a TROM object, can be captured in a statechart diagram. The port, enabling, and post conditions of a transition specification can be expressed as Boolean expressions associated with transitions between states in the statechart diagram. Timing constraints on reactions to stimuli can be captured in a statechart diagram, where the constraints are specified as Boolean expressions involving logical clocks. In a sequence diagram, identifiers associated with occurrences of events can be used in expressing timing constraints.

The configuration of a subsystem with instances of TROM classes can be described in a UML collaboration diagram. A subsystem included in another subsystem can be described by a distinct collaboration diagram; however, an interaction involving objects from different subsystems need to be described by a collaboration diagram incorporating the objects participating in the interaction. A collaboration diagram also captures the configuration of a subsystem in specifying which objects interact with an object, and consequently, which messages are transmitted in the interaction.

2.4 PVS – Prototype Verification System

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 be used to detect flaws in the design at an early stage, thus reducing the costs of remedying faulty systems.

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.

The Choice of PVS - Justification

We need to describe the semantics of UML notation in a formal way to allow precise description and analysis of the behavior of objects and subsystems. PVS specification language suits the requirements of a notation in which well-formedness rules, invariants, and constraints can be precisely specified. UML semantics described in PVS specification language serves as both a communication mechanism and a proof mechanism.

There is an increasing demand on the construction of provably correct software systems in strategically important areas, such as the aerospace industry and NASA projects. The current status of formal methods integration in industrial-strength software development includes applications in areas such as avionics, telecommunications and nuclear power plants. PVS is being explored as a tool for specification and verification for mission-critical software in NASA projects [LA94]. Experience gained from these studies are reported in two NASA guidebooks [NAS95] [NAS97]. Easterbrook et al. [ELC+98] give an extensive experience report on the use of PVS for design analysis in NASA projects, and how PVS is being groomed for use in the integration of formal methods in the development process of safety-critical systems.

The higher-order logic of PVS, together with its rich and rigorous type system, brings lot of expressive power to the specification language. This makes PVS suitable for formally describing semantics of complex structures, as those used for the abstract syntax and well-formedness rules of UML notation. Moreover, PVS supports the specification of abstract data types in a concise and efficient way, with the automatic generation of axioms and functions capturing the intended properties of the data types. Another aspect of PVS is its powerful reasoning system. It supports a wide range of decision procedures, provides an extensive set of proof commands, and supports interactive proof construction. These features make PVS well-suited for stating time-dependent properties and verifying their presence in design specifications. Shankar [Sha93] gives a theory of time, and a computational model for specification and verification of real-time systems.

PVS Specification Language

PVS specification language allows abstract data types, programs, specifications, axioms, lemmas and theorems to be described within theories. Theories can be parameterized, and an *Import* clause allows theories and instances of parameterized theories to be included in other theories. PVS prelude includes a rich set of theories and abstract data types capturing a wide range of mathematical models. PVS augments classical higher-order logic with

- a sophisticated type system containing predicate subtypes and dependent types,

- parameterized theories, and
- a mechanism for defining abstract data types in a concise way.

Standard PVS types include numbers, records, tuples, arrays, functions, sets, sequences, lists, and trees. This combination of features in the PVS type system is convenient for specification, but makes type-checking undecidable. The PVS type-checker copes with this undecidability by generating proof obligations for the PVS theorem prover; most of these proof obligations can be discharged automatically. Moreover, this mechanism allows PVS to enforce strong checks on consistency and other properties.

PVS Theorem Prover

Theorem proving in PVS is based on *sequent calculus*; a sequent consists of a set of antecedent formulas and a set of consequent formulas. In proving a theorem, we demonstrate that the conjunction of the *antecedents* implies the disjunction of the *consequents*. PVS includes an extensive set of proof commands, classified as *primitive rules*, *defined rules*, and *strategies*. The logic of PVS is described in terms of inference rules; the *primitive rules* are derived mainly from these inference rules. The defined rules are obtained by composing primitive rules into more complex commands. Strategy constructors allow the user to define strategies based on patterns of inference steps. In addition to the defined rules and strategies provided, user-defined proof commands can be included in the system.

PVS is a powerful interactive theorem prover/proof checker. Its basic deductive steps are based on atomic commands for

- induction,
- quantifier reasoning,
- simplification using arithmetic and equality decision procedures and type information,
- propositional simplification using binary decision diagrams, and
- automatic conditional rewriting.

The proof construction process is managed by prompting the user for a suitable command for a given subgoal. Execution of a command can generate further subgoals, or complete a subgoal and move the control over to the next subgoal in a proof. User-defined proof strategies can be used to enhance the automation in the proof checker. Model-checking capabilities used for automatically verifying temporal properties of finite-state systems have been integrated into PVS.

Completeness in Datatype Specifications

PVS provides a powerful mechanism for defining abstract data types. A PVS data type is specified by providing a set of *constructors* along with associated *accessors* and *recognizers*. A data type declaration is of the form

$\mathtt{adt} : \mathtt{DATATYPE}$
$\mathtt{BEGIN}$
$\mathtt{cons}_1(\mathtt{acc}_{11} : \mathtt{T}_{11}, \ldots, \mathtt{acc}_{1n_1} : \mathtt{T}_{1n_1}) : \mathtt{rec}_1$
$\vdots$
$\mathtt{cons}_m(\mathtt{acc}_{m1} : \mathtt{T}_{m1}, \ldots, \mathtt{acc}_{mn_m} : \mathtt{T}_{mn_m}) : \mathtt{rec}_m$
$\mathtt{END\ adt}$

where the $\mathtt{cons}_i$ are the constructors, the $\mathtt{acc}_{ij}$ are the accessors, the $\mathtt{T}_{ij}$ are type expressions, and the $\mathtt{rec}_i$ are recognizers. Each line is referred to as a *constructor specification.* A specification for the *stack* data type is

```
stack [t : TYPE] : DATATYPE
 BEGIN
  empty : emptystack?
  push(top : t, pop : stack) : nonemptystack?
 END stack
```

In this specification, empty and push are *constructors*, top and pop are *accessors*, and empty-stack? and nonemptystack? are *recognizers* of the parameterized stack type.

When a data type is typechecked, a new theory is created that provides the axioms and induction principles needed to ensure that the data type is the initial algebra defined by the constructors. The generated axioms relate the constructors, accessors and recognizers introduced for the abstract data type. For instance, *extensionality* and *eta* axioms are generated to define equality on instances of the abstract data type. Other axioms are included to define well-foundedness rules, and to support well-founded subterm ordering relations and strong forms of induction. The functions *every* and *some* are generated to establish the truth value of a predicate in existentially and universally quantified formulas on the data type. Two functions are included to define a generic *map* on the abstract data type. These generated axioms and functions must often be manually augmented with additional axioms and function definitions capturing other properties of the data type to ensure *completeness* of the data type specification. A data type specification is *complete* if every intended property of the data type can be deduced from the axioms.

The rich type system of PVS is supported by the generation of proof obligations, called *type constraint conditions (TCC's)*, which can be discharged by the proof checker. In proving a theorem, subproofs may require discharging some of these obligations. This can often be done by invoking the predicates used in subtyping, the axioms generated for abstract data type definitions, and user-defined axioms capturing additional properties of these data types. Whenever a proof cannot be discharged, it is due to the incompleteness of the specifications. To remedy this situation, we need to include more properties in the data type specifications.

Chapter 3

Literature Survey

Several notations have been proposed for modeling real-time reactive systems in an object-oriented style. However, since several of them are mostly object-based, few actually reap the full benefits of object-orientation. Since the inception of UML, several researchers have been working either on its formalization, on providing formal semantics for the notation, or on both in parallel. In most cases, the work focused on a subset of the notation, and ignored the other kinds of diagrams. Formal verification has been proposed for safety and liveness properties in the context of safety-critical systems. The two main approaches to verification are model-theoretic where a given temporal formula is applied to the constructed model, and proof-theoretic reasoning where logical deductions are applied to demonstrate that a lemma is a logical consequence of a set of axioms.

3.1 Introduction

This chapter describes work related to visual modeling of real-time systems, formal semantics for object modeling techniques, and formal verification of real-time systems. Section 3.2 surveys work related to object-oriented modeling of real-time reactive systems. We study different modeling techniques, and draw a comparative analysis with the proposed methodology. Section 3.3 provides an overview of research related to formalization of UML, defining formal semantics for the notation, and for object modeling techniques. Section 3.4 surveys work related to methodologies for formal verification of safety and liveness properties in the design real-time reactive systems.

3.2 Modeling Real-Time Reactive Systems

Douglas [Dou98a, Dou98b] surveys the advantages of applying object-orientation in modeling real-time systems. These include improved problem domain abstraction, improved scalability, better support for reliability and safety concerns, and inherent support for concurrency. In bringing out the importance of message properties in requirements analysis of real-time systems, Douglas describes how arrival patterns and synchronization patterns can be combined in specifying message passing semantics in UML. Analyzing message properties is crucial in defining timing requirements of real-time systems in terms of response time, throughput, service time and latency. Selic, Gullekson and Ward [SGW94] identify the following characteristics of application domains relevant to real-time systems: timeliness, dynamic internal structure, reactiveness, concurrency, and distribution. The authors sustain that object-orientation is a suitable abstraction for dealing with the inherent complexity, by defining an object as a *logical machine* modeling a component of a system.

Selic and Rumbaugh [SR98] discuss an embedding of ROOM (Real-Time Object-Oriented Modeling) [SGW94] methodology in UML. ROOM is based on *actors* [Agh86], encapsulated concurrent objects communicating through point-to-point links between interface objects called *ports*. A *message* consists of a signal name, data, and a priority level. Ports represent *protocols* defining sets of incoming and outgoing messages. *Bindings* establish message channels between actors, constraining communication relationships. A major drawback of ROOM is its restricted applicability to input-enabled systems [LT87]; however, ROOM supports certain features of object-orientation. In spite of the apparent closeness of our model to ROOM, there are significant differences.

- ROOM supports two types of timing constraints: *latency*, and *service* times. Both relate to implementation resources; timing constraints on stimulus-response behavior cannot be specified. In our approach, any type of timing constraint associated with a state transition can be specified.
- Every transition in a ROOMchart [SGW94] requires an implementation so that the behavior of the model can be observed; operation descriptions use concrete state variables. This brings forth a bias towards an implementation in the early design stages, reducing the scope for design validation and refinement of the model. Our approach emphasizes an abstract specification of transitions, with several forms of nondeterminism.
- ROOM does not support data abstraction. In our methodology, reactive object models incorporate abstract data types.

- ROOM lacks a formal semantics, and consequently, rigorous validation and verification methods are not available. In our approach, we develop a logical semantics for UML design models as a stepping stone for conducting formal verification.

Recent works applying the object paradigm for modeling reactive systems include DisCo [JKSSS90]. The significant differences between DisCo and our methodology follow.

- Although the claim is that DisCo specifications are object-oriented, they are only object-based and do not have the expressivity of UML notation. Our work extends UML notation for modeling reactive objects, and derives the benefits of object-orientation.

- The notion of time is not included in DisCo models, and real-time constraints cannot be specified. We extend UML notation to capture timing constraints on the behavior of reactive objects.

- In DisCo, time-dependent properties cannot be stated as part of requirements specifications. In our work, both time-dependent and invariant properties can be specified and verified.

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 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.

Other works related to state-based modeling of reactive systems are Objchart [GM93], and TRIO++ [MSP94]. Objchart do not support specification of real-time constraints. TRIO++ emphasizes expressing the requirements specification of real-time systems using object-oriented principles. However, the language lacks important object-oriented 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.

Several tools have been introduced for the development of reactive systems. These include 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 TROMLAB environment differs from the above two tools in several important ways:

- TROMLAB 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.

- RTUML notation, which is used as a front-end in TROMLAB, has a formal operational semantics, supporting rigorous analysis techniques.

- The design specification technique in TROMLAB is three-tiered: the first tier specifies data abstractions in *Larch Shared Language* [GH93]; the second tier provides the specification of the classes of objects in the problem domain; and the third tier constitutes *System Configuration Specifications*, describing object interactions.

3.3 Formal Semantics

We survey work related to defining semantics for visual modeling languages, with emphasis on the UML notation. We first review the semantics document provided with the specification of the language [OMG99], and include a critique of the use of OCL for its specification. We subsequently review work related to formalizing UML and providing formal semantics for the notation. We include works involved in providing formal semantics for visual object models and message sequence charts.

3.3.1 UML Formalization and Formal Semantics

The UML Semantics document [OMG99] describes a metamodel used to specify the abstract syntax and semantics of UML object modeling concepts. It uses UML class diagrams to describe the abstract syntax, OCL (Object Constraint Language) [OMG99] to describe the static semantics in the form of well-formedness rules on UML model elements, and natural language text and diagrams to describe the dynamic semantics. It is written in a semi-formal style, using a formal language to complement a natural language and a graphic notation. The document specifies semantics for both structural and behavioral object models. It gives three views of the metamodel, in the form of an abstract syntax, well-formedness rules, and informal semantics.

The UML metamodel is organized into logical packages grouping metaclasses. The structure of the organization is done in such a way that there is strong cohesion within each package, and loose coupling among metaclasses in different packages. A small subset of the UML notation is used to express the abstract syntax for the UML semantics. The well-formedness rules are described using OCL, and the semantics capture the meaning of model elements in natural language text and diagrams. The mapping of the UML graphic notation to the underlying semantics is described in the UML Notation Guide [OMG99]. The consequence of this exercise is an ambiguous communication mechanism in which behavioral analysis is unreliable.

The use of a metamodel to describe itself brings about certain theoretical limitations; however, this is compensated by more expressiveness and readability. The authors argue that a completely formal specification of UML would add significant complexity without clear benefit. They also argue that the state of practice in formal specifications does not yet address some of the more difficult language issues that UML introduces. The thesis proposes a modeling technique based on a subset of UML notation augmented with minimal extensions and well-formedness rules, and provides a formal semantics for the hybrid notation as a foundation for design analysis and formal verification.

Abstract Syntax

The syntax of a language defines what constructs exist in the language, and how the constructs are built up in terms of other constructs. The syntax of UML is defined independent of any notation and is therefore abstract. A subset of the UML notation consisting of a UML class diagram and a supporting natural language description are used to describe the abstract syntax. By mapping the graphic notation of UML onto the abstract syntax, we obtain its concrete syntax; this mapping is described in the UML Notation Guide.

A class diagram presents the abstract syntax for each UML logical package. The diagram shows the metaclasses defining the constructs and their relationships, together with some of the well-formedness rules. The rules included in the diagram relate mainly to the multiplicity requirements of the relationships, and whether or not the instances of a construct must be ordered. A natural language description is included for each construct; this description presents the construct and defines the metaclass specifying the construct, enumerating its attributes and associations.

Static Semantics - Well-formedness Rules

The static semantics of a language defines the ways in which an instance of a construct of the language should be connected to other construct instances to be meaningful. The static semantics is given in the form of rules for the well-formedness of a description in the language. The well-formedness rules of UML are described using both a formal language (OCL) and natural language text.

Other than those relating to multiplicity and ordering constraints, well-formedness rules for each construct are defined as a set of invariants on an instance of the metaclass specifying the construct. These rules pertain to constraints over attributes and associations defined in the metamodel. An invariant is defined in terms of an OCL expression; the expression is annotated with an informal description. In some cases, the rules already defined in superclasses together with the rules included in the class diagrams are sufficient to express the static semantics of a construct.

Dynamic Semantics

The meaning of a well-formed construct of a language is defined by the dynamic semantics of the language. The dynamic semantics of UML are described in natural language mainly. Sometimes an additional notation pertinent to the construct being described is used to complement the English text. The dynamic semantics of only concrete metaclasses are defined; abstract metaclasses do not have a meaning in the language. Semantics are defined for constructs grouped into logical chunks.

Specification of UML Using OCL

OCL (Object Constraint Language) is a typed functional language supporting first-order formulas for specifying relationships and constraints on objects. Warmer et al. [WHCS97] describe their experience with using OCL to specify CMM (Common Metamodel) and UML submissions to the OMG (Object Management Group). We give a brief description of OCL, and a critique of the issues brought up in the use of OCL to specify UML. OCL is a simple language based on first-order predicate logic, for specifying properties. OCL has a set of basic types, with a set of basic operations on these types. Constraints are expressed as Boolean expressions, and apply to a collection of elements. The core language includes the collections Set, Bag, and Sequence. Expressions can be built on universally or existentially quantified formulas, using the *forall* and *exists* constructs, and set operations, such as *union*. A query can be formulated as an expression returning a collection; this expression can be used in other expressions. Parameterized operations are allowed; these can be used to express recursive constraints, and in more complex expressions.

OCL is a pure expression language; expressions do not have side effects, and therefore do not change the state of the system. However, the post-condition of an operation denoting a state change can be specified using an OCL expression. The evaluation of an OCL expression yields a value. The logic or control flow of a program cannot be described in OCL, and expressions may not be executable; implementation issues cannot be expressed. OCL can be used for specifying (i) invariants on types and classes, (ii) pre and post conditions on operations and methods, (iii) guards, (iv) constraints on operations, and (v) definitions of operations.

Issues in Using OCL to Specify UML - A Critique

Warmer et al. [WHCS97] raise the question of how much to specify in an attempt at giving a precise description of the model elements of UML. They relate this issue to the difference between a communication mechanism and a proof mechanism, arguing that whatever is enough to satisfy the reader is sufficient. We note that a proof mechanism not only demonstrates the satisfaction of properties, but also convincingly communicates aspects of the behavior of a system. An *implies* clause capturing the intended consequences of a theorem can be used to establish that properties stated in requirements specifications are captured by design specifications. The proofs of claims in the implies clause strengthen the confidence in the analysis process and in developing a correct design. We conclude that a proof mechanism is a communication mechanism, and that communication mechanisms can be at different levels of formality.

Another issue raised is the use of tools to create good designs, arguing that such tools can only identify bad designs, and cannot lead a user in the construction of a good design. We remark that the identification of bad designs is essential in the path to the development of a good design. Constructing a correct design involves ascertaining that properties stated in requirements specifications are present in design specifications. This often involves an iterative design approach, where the model is repeatedly modified until all the requirements are incorporated. The design analysis process can be supported by appropriate tools, such as animation tools for executing design specifications to observe system behavior, and theorem provers for establishing that specifications meet specific properties. The third issue raised is the dependence of OCL on the system being modeled. This aspect is unsafe in that it can give rise to paradoxes analogous to the set inclusion paradox. This dependence of OCL contrasts with the attributes of formal specification languages whose notations do not have any dependency on the system being modeled. One possible approach is to liberate OCL from this constraint.

3.3.2 Techniques for Defining Formal Semantics

Rumpe [Rum98] clarifies the concepts of syntax and semantics and their relationships, with emphasis on the formalization of UML and the semantics of its constructs. In surveying work on UML formalization, the author points out the implicit assumptions made in different approaches, such as the subset of the notation formalized, the assumptions on the application domain, the relationship defined between the syntactic constructs and the notion of a system, and the formalization technique. The author argues that when used for a semantics definition, a notation should not be defined using the same notation, but in a notation that has a semantics.

Context conditions, in the form of well-formedness rules, constrain the syntax for correctness, but do not provide a meaning to the syntax. Context conditions result in a language to which a sound semantics can be applied, that is, a meaning can be given to every well-formed construct in the language. A metamodel defines the abstract syntax of a language; a metamodel is relation-based and does not have a canonical point where to start the semantic definition.

A semantic definition consists of a semantic domain and a mapping from the syntax to the semantic domain. The meaning of each construct of the language is given in terms of concepts in the semantic domain. The semantic domain reflects the concepts that exist in the universe of discourse, and is usually called the *system model*, as it describes the conceived notion of what a system is. The semantic domain is an abstraction of reality, describing the important aspects of the kind of system being considered. The explicit definition of the semantic domain allows to understand the application domain. It is important to distinguish semantics, which implies *meaning*, from *behavior*, which is represented in terms of syntactic constructs in a similar way as *structure* is described.

A semantic mapping relates the syntactic constructs with the concepts in the semantic domain. The explicit definition of the mapping allows reasoning about it. A mapping can be defined in an algorithmic fashion and implemented. This allows the translation of a description in the language into a description in terms of the semantic domain concepts, and subsequent use of proof and analysis techniques. The author applies these semantic definition principles to a hierarchical Mealy Automata for illustration.

One of the goals of defining semantics is to produce an improved version of the notation, and to gain insight into it. For instance, sufficient context conditions constraining the notation may be provided to ensure consistency of the resulting semantics. A semantic definition is intended for either the notation developer, the tool vendor, or the user of a language. The notation developer uses the semantic definition to gather insight into the syntax of the language. The semantic definition provides a user of the language with intuition into the purpose of the notation. The tool vendor uses the semantic definition to determine how the symbols of the notation can be modified and how code can be generated.

Bourdeau and Cheng [BC95] provide a formal semantics for object model diagrams in OMT (Object Modeling Technique) [RBP+91]. The result is a method for deriving modular algebraic specifications directly from object model diagrams. The formalization of the object models provide a basis for deriving system designs. The authors make use of instance diagrams in defining the semantics of an object model. The state space of an object model is defined as the set of all instance diagrams of that object model. An object model diagram corresponds to an algebraic specification, and an instance diagram of the object model corresponds to an algebra that satisfies the algebraic specification.

Mauw and Reniers [MR94] provide a formal semantics of *basic message sequence charts* based on process algebra. The authors justify this choice with the argument that all features incorporated in the theory of message sequence charts are related to topics in process algebra, such as the state operator and the global renaming operator. Message sequence charts provide a graphic notation for describing interaction between system components, and is applied mainly in telecommunication systems. A message sequence chart expresses an execution trace of the behavior of a system. A collection of message sequence charts provides a detailed specification of a system.

The Precise UML group includes several researchers who have initiated efforts in view of developing UML as a formal modeling notation [EBF+98, EFLR98, FEKB98, Lan98]. Other works in formalizing UML include [BHH+97, CE97, ECM+99]. The motivation comes from the lack of a precise semantics for the modeling language. The interpretation given to the meaning of the diagrams is imprecise and results in confusion. Design analysis cannot be conducted in a rigorous manner, and the consistency of implementations and models cannot be established. The group aims at investigating the completeness of the semantics [OMG99] and at developing approaches for using UML precisely, including formal development methods, and analysis and refinement techniques.

Some of the approaches taken within the group include the following.

- Formalize UML interaction diagrams using a Real-Time Action Logic, and develop methods for verification of refined models against abstract models.
- Elaborate the notion of refinement and composition for UML diagrams.
- Integrate UML with a mature formal specification notation, such as Z.
- Develop a deductive system for UML which can be used to verify properties.

3.4 Formal Verification

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. Analysis techniques for real-time reactive systems are mainly adaptations of techniques originally developed for non-real-time systems. The major difference is in the emphasis on the time dependency. It is imperative that a property for a real-time reactive system holds at specified times for all environmental situations. Verification of such properties demand identifying invariant properties for the system and rigorously proving that they are present in the design. The two main categories of verification approaches are *model checking* and *proof-theoretic reasoning*.

Model Checking

Model checking is a powerful technique for the automatic verification of discrete finite-state systems, real-time systems, and hybrid systems. Since its inception by Clarke and Emerson [CE81], model checking has been successfully applied to verify hardware circuits and communication protocols [LP85, RSV87, McM93]. More recently, model checking tools have been developed for real-time systems [MSJ96], and hybrid systems [HH95]. A model checker is an algorithm which determines whether or not a mathematical model of a system satisfies a requirement specified as a temporal logic formula. The mathematical model is a finite state machine for discrete finite-state systems, and is in general infinite for real-time and hybrid systems.

Although model checkers have been applied to verify large systems, for models with more than 10^{20} states, their application to real-time systems still remains largely in the research domain. Recently techniques have been developed to handle discrete as well as dense time models. Alur and Dill [AD96] have extended the theory of finite automata to incorporate time and have introduced heuristics to alleviate exponential state explosion in the search space towards a verification of real-time systems. The automata-theoretic approach to verification is based on model checking and can handle systems with one train, one gate, and one controller. This complexity is also demonstrated by the Modechart verifier [MSJ96], and the model-checking tool for hybrid systems implemented in *HYTECH* [AHH96]. The main drawback in using model checkers for real-time systems is one of scale; the addition of continuous real-value variables such as time, pressure, and temperature, produce a computation model that is too large to analyze. The *HYTECH* prototype works efficiently and has been applied to verify safety properties in several problems, including the railroad controller involving one train, one gate and one controller.

Proof-Theoretic Reasoning

A model checker automatically generates the results by applying the given temporal formula to the constructed model. In contrast, in the proof-theoretic approach interaction is often required to apply logical deductions. The system design is described as a set of axioms, and the property to be verified in the system is stated as a lemma to be proved. The verification procedure constructs a proof, often mechanically and interactively, for the lemma as a consequence of the axioms. Boyer-Moore theorem prover [BM88], EVES [KPS⁺93], LP [GH93], HOL [Gor88], and PVS [ORSvH95] are some of the theorem provers that are used to mechanically construct proofs, to check hand-written proof steps, or to interactively construct a proof for an axiomatized system using the proof commands and strategies available in the tool.

The proof proposed by Shankar [Sha92] applies the sequent calculus approach to theorem proving on the theory of the railroad crossing system expressed in PVS specification language. The methodology includes a computational model for system behavior and a theory of time; it uses the *since* operator to describe durations on state predicates. The methodology was applied to construct a proof, using PVS theorem prover, for the safety property of the railroad crossing. The proof involves formulating a set of invariance assertions and proving them in the model of program behavior consisting of transition actions and time constraint axioms.

In Shankar's computational model [Sha93], 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 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 given by the specification. 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 a 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; 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.

Heitmeyer and Lynch [HL96] have applied a proof-theoretic approach to show the correctness of a railroad crossing system implementation against its operational specification. The specification method is based on timed automata, and proofs are constructed manually. The verification method uses reachability analysis and induction on the admissible execution sequences. Archer and Heitmeyer [AH96a] have developed a template containing a set of common theories and a common structure in PVS specification language for constructing timed automata models and proving properties about them using PVS proof checker. The authors provide a proof for the safety property of the railroad crossing system based on state invariants. The proof applies the induction principle on system states to demonstrate that the property is true in all reachable states of the model. Archer and Heitmeyer [AH96b] propose a system providing support for producing specifications and verifying proofs based on the PVS-based methodology.

Kellomaki [Kel97a, Kel97b] describes a mechanical verification support for the DisCo specification language [JKSSS90]. The author describes the mapping of the DisCo language onto PVS higher-order logic, and the use of the theorem prover for verifying two invariant properties of the alternating bit protocol. The methodology applies to verifying invariants at an abstract level of design specification. In refinement steps, concrete variables introduced to implement the abstraction lead to proof obligations to demonstrate how the values of the abstract variables can be computed from the concrete variables. Time-dependent constraints and properties cannot be stated as part of requirements specifications in the DisCo language.

Lutz and Ampo [LA94] give an extensive experience report on requirements analysis based on a methodology incorporating OMT (Object Modeling Technique) and PVS (Prototype Verification System). This approach does not integrate the graphic notation of OMT with the formal specification language of PVS; it merely uses these notations as complements to each other. The authors argue that such an approach allows a smooth progression to constructing a correct design. While the OMT model provides a high level structural view of the requirements, the PVS model gives a more detailed view and supports a detailed behavioral analysis. However, it is not apparent how correspondence is established between the OMT diagrams and the formal specifications in PVS for each model element.

In the approach adopted by Lutz and Ampo, the initial step is to model the requirements of a system in the semi-formal graphic notation of OMT. The purpose of this exercise is to obtain insights into the initial design, detecting modeling flaws early in the design process. The authors report that the use of intermediate structured representations in the form of OMT diagrams help defining the boundaries and interfaces of the system. This is performed at an early stage in the development process, while the requirements are being compiled and are therefore still in a volatile state. The diagrams help to understand the system, and reduce the effort needed in building the formal specifications.

Once an initial design is obtained in OMT diagrams, the requirements of the system are specified in the PVS specification language. This allows a more rigorous analysis of the model since the PVS theorem prover can be used to establish the inherence of certain properties in the design. The PVS model is derived from the OMT model: elements of the OMT model are mapped onto elements of the PVS model. For instance, classes are mapped onto type definitions, and state transitions are mapped onto functions and axioms. The PVS model is then used to establish the satisfaction of safety and liveness properties.

Our approach to verification of real-time reactive systems is founded on an object-oriented framework, and uses the PVS specification and verification system as a back-end. We have developed methods (i) to generate object-oriented formal specifications from UML models, (ii) to map the design specifications onto PVS theories containing axioms capturing the behavior of classes of reactive objects and subsystem configurations, and (iii) to prove a safety property formulated as a theorem within the theory describing the overall system. We applied the methodology to mechanically construct a proof for the safety property of the generalized railroad crossing system, with an arbitrary number of trains, gates, and controllers, and with non-overlapping parallel tracks through a crossing. The proof steps for this version of the railroad crossing system can be defined as a strategy and applied to mechanically verify other models of the railroad crossing system. In our methodology, both time-dependent and invariant properties can be specified and verified.

Chapter 4

RTUML - Real-Time Unified Modeling Language

Applying UML extension mechanisms, we develop a visual modeling technique based on the abstract object model for real-time reactive systems. While the extensions are minimal, with stereotypes on classes and associations, the resulting notation is expressive enough to model the structure and behavior of concurrent communicating objects and real-time systems. To provide a formal framework for rigorous analysis of UML design models, we devise a translation mechanism to derive formal specifications from the UML description of a reactive system, making the formal notation transparent to the designer. We provide logical semantics for UML reactive system models in the object constraint language OCL enriched with temporal predicates.

4.1 Introduction

This chapter is organized as follows. Section 4.2 outlines the UML extensions introduced to support the modeling technique for real-time reactive systems. Section 4.3 enumerates the extensions to UML notation, and the restrictions on UML notation in terms of well-formedness rules for RTUML. It includes a specification of the RTUML well-formedness rules in OCL. Section 4.4 uses a distributed navigation control system for illustration of the visual modeling technique. Section 4.5 describes the mapping of a RTUML reactive system model onto formal specifications, and illustrates the translation process with the case study. We also sketch how the formal specifications can be used for validation and verification purposes.

4.2 Extending UML to Support the Real-Time Reactive Model

In developing a visual modeling technique based on the abstract real-time reactive model, we apply the extension mechanisms of UML to introduce minimal extensions to the notation. We maintain the distinction between the three layers of the abstract model, namely the data models, the generic reactive object model, and the subsystem model. The purpose of providing a graphic model of a reactive system is to serve as the front-end of a design tool. Although rigorous analysis is not attainable at this level, visualizing the entities and the configuration of a system provides insight into its overall structure and behavior. Translating the graphic model of a reactive system into formal specifications alleviates the need for a system designer to learn and use the formal specification language.

Modeling a reactive system within the proposed framework involves describing generic reactive classes, and relationships among collaborating objects. Entities may require abstract data structures to support their functional behavior. Types of relationships among objects and classes include associations, aggregations, composition aggregations, and specializations. UML *static structure* diagrams support the specification of such entities and relationships. The two types of interaction that are relevant to the design of reactive systems are sequential composition and concurrency. UML *collaboration* diagrams depict collaborations among objects, showing the context for the purpose of the cooperation. Superimposing interactions on the collaborations capture sequences of messages exchanged among the objects.

4.2.1 Visual Model of Reactive System

Applying UML extension mechanisms, we introduce two new class stereotypes: (i) *GRC (Generic Reactive Class)*, to capture a generic reactive object model, and (ii) *PortType*, to define a port type and its associated events. Composition aggregation relationships associate PortType classes with a GRC class. We use the UML classifier *DataType* to describe LSL traits specifying abstract data structures. This classifier is appropriate for specifying a trait, since in the modeling stage we focus on the operations defined on the data structure, and disregard the axioms specifying its behavioral properties; these axioms are relevant in the analysis and verification stages. The *attribute compartment* of a GRC class includes the attributes of a reactive object model, that is, instances of either a PortType class or a DataType. The structure of instances of a GRC class includes ports as instances of PortType classes, and attributes as instances of PortType or DataType classes. In a class diagram including more than one GRC class, the stereotype *PortLink* on binary associations between port types defines the validity of communication channels.

To describe the functional and temporal behavior of a generic object model, we associate a UML *statechart* diagram with each GRC class. The label on a transition in a statechart diagram describes the triggering event, and logical assertions conditioning the occurrence of the event. The *guard* of a transition includes the assertions corresponding to the port condition and the enabling condition, and the expressions involving logical clocks specifying the time window for the constrained reaction; a negative value for a logical clock indicates a disabling state for the time constraint. The *action* of a transition includes the assertion corresponding to the postcondition, and the expressions initializing the logical clocks defining time constraints on a reaction.

We use a UML *collaboration* diagram to model the configuration of a subsystem by describing links between specific ports of instances of the generic reactive classes. We introduce a new stereotype *PortLink* on binary associations between port objects, to define links between reactive objects. Associations between ports indicate links for communication, consistent with the synchronized events in the corresponding statechart diagrams. Messages included in an interaction superimposed on the collaboration correspond to synchronous transitions in the statechart diagrams. The timing of the event occurrences must be consistent with the time constraints in the statechart diagrams.

A UML *sequence* diagram describes an interaction scenario conforming to the configuration captured in a collaboration diagram. A scenario is either specific with time constants, or general with time variables for event occurrences. A specific scenario describes a sequence of events occurring at pre-determined times. A general scenario is a visual illustration of the time constraints imposed on system behavior. In a general scenario, linear inequalities involving the logical variables describe time constraints. A scenario describes a sequence of events occurring at pre-determined times in conformance with the port links between reactive objects for synchronous message passing. Linear inequalities involving logical variables over absolute time describe time constraints in a sequence diagram. Each time constraint specifies a window during which an object can react by firing either an internal transition, or an output message. A valid scenario must comply with the time constraints described in the corresponding statechart diagrams. Since our goal is to develop a verification methodology to formally verify whether a property holds in all scenarios adhering to the time constraints, we use a sequence diagram with absolute time variables for event occurrences.

4.3 RTUML Abstract Syntax and Well-formedness Rules

We apply UML extension mechanisms to introduce (i) stereotypes on classes to model generic reactive classes and port types, and (ii) stereotypes on associations to model the composition aggregation relationship between a GRC class and its port types, and the relation between port types denoting communication channels. The following enumeration of new stereotypes corresponds to the extensions in UML Package *Foundation*:

- UML Metaclass *Class*
    1. New stereotypes on *Class* model element
        (a) *GRC*
        (b) *PortType*

- UML Metaclass *Association*
  1. New stereotypes on *Association* model element
     (a) *PortAggregation*
     (b) *PortLink*

4.3.1 Restrictions on UML Notation

In this section, we provide an informal listing of the restrictions on UML notation in terms of well-formedness rules for RTUML. We classify these rules in terms of the corresponding UML model elements.

- Classes in Class Diagrams
  - Class
    1. Only classes with the stereotypes *GRC* and *PortType* are allowed.
    2. Inheritance is not currently supported.
    3. A class can only contain attributes.
  - GRC Class
    1. A reactive object maintains its own single thread of control.
    2. A GRC class associates with only PortType classes.
    3. A GRC class associates with at least one PortType class.
    4. Each instance of a GRC class holds its own value of the attributes of the class.
    5. Each instance of a GRC class holds only one value of the attributes of the class.
    6. An attribute of a GRC class is not visible outside the class.
    7. Each instance of a GRC class can modify the value of its attributes.
    8. An attribute of a GRC class is an instance of either a PortType, a DataType, or a binding of a template DataType.
    9. An attribute of a GRC class can be an instance of a PortType only if the GRC class aggregates the PortType.
    10. If an attribute of a GRC class is an instance of a binding of a template DataType, then every actual argument of the binding is either a DataType or a PortType.
    11. If an attribute of a GRC class is an instance of a binding of a template DataType, and an actual argument of the binding is a PortType, then the GRC class aggregates the PortType.
  - PortType Class
    1. A port object does not maintain a thread of control.
    2. A PortType is owned by exactly one GRC class.
    3. A PortType is linked to at least one other PortType.
    4. A PortType has a single attribute.
    5. The value of the attribute of a PortType is common to all instances of the PortType.

6. Only one value of the attribute of a PortType is maintained.
7. The value of the attribute of a PortType is visible from outside the class.
8. The initial value of the attribute of a PortType cannot be modified. The identifier of the attribute is "events", and the identifier of the type of the attribute is "Set".

- Associations in Class Diagrams
  - Association
    1. Only associations with the stereotypes *PortAggregation* and *PortLink* are allowed.
    2. All associations are binary associations.
    3. The ends of an association can be used to access the target reactive or port object.
    4. There is no ordering in the ends of an association.
    5. The ends of an association are connected to instances of classes.
    6. Each end of an association is connected to only one instance of a class.
    7. The ends of an association cannot be modified.
    8. An end of an association is visible from the opposite end of the association.
  - PortAggregation Association
    1. A GRC class is a composite of its PortTypes.
  - PortLink Association
    1. PortTypes are connected by associations with no aggregation relationship.
- Interaction Diagrams – Collaboration Diagrams and Sequence Diagrams
  - Collaboration
    1. Only instances of GRC classes and PortTypes are allowed in a collaboration.
    2. Every instance of a GRC class is connected to at least one instance of one of its PortTypes.
    3. Every instance of a GRC class is connected to at least one instance of one of its PortTypes, where the PortType instance is linked to another PortType instance.
    4. Every instance of a PortType is connected to exactly one instance of a GRC class.
    5. Every instance of a PortType is connected to at most one instance of a PortType.
  - Classifier Role
    1. Each object in a collaboration specifies a single instance of a class.
    2. Each object in a collaboration specifies an instance of either a GRC class or a PortType.
  - Association Role
    1. Each connection in a collaboration specifies a single link.
    2. Each connection in a collaboration specifies a link corresponding to either a PortAggregation Association or a PortLink Association.
    3. Each connection in a collaboration connects exactly two objects.

4. In a collaboration, a connection that links an instance of a GRC class and an instance of a PortType, corresponds to a PortAggregation association between the GRC class and the PortType.
5. In a collaboration, a connection that links an instance of a PortType and an instance of another PortType, corresponds to a PortLink association between the two PortTypes.

- Interaction
  1. The sender of every message in an Interaction is an instance of a GRC class.
  2. The receiver of every message in an Interaction is an instance of a GRC class.
  3. The communication connection for every message in an Interaction corresponds to an association between two PortType instances, where the PortType instances are part of the sender and receiver objects of the message.

• Statechart Diagrams

- State Machine
  1. A statechart represents the behavior of instances of a GRC class.
- Transition
  1. The source and target of a transition is a state.
  2. A transition has a trigger event.
  3. A transition has a guard condition.
  4. A transition has an effect action.
  5. The trigger event of a transition is a signal event.
  6. Parameterized events are not currently supported.
  7. The effect action of a transition occurs instantaneously.
- State
  1. A state is either simple or complex.
  2. A state has no exit action.
  3. No activity takes place while being in a state.
  4. No event is retained in a state for further consumption.
- Complex State
  1. A complex state is not concurrent.
  2. Every subvertex of a complex state is a state.

4.3.2 RTUML Well-formedness Rules in OCL

This section gives a specification of the RTUML well-formedness rules in OCL. Each rule has a corresponding informal description in the previous section.

UML Package *Foundation*

Subpackage *Core - Backbone*

- Class

1. A Class has either the stereotype <<GRC>> or the stereotype <<PortType>>.

```
self.stereotype.name = ''grc'' or
    self.stereotype.name = ''porttype''
```

2. A Class is a root, is a leaf, and is not abstract.

```
self.isRoot and self.isLeaf and not self.isAbstract
```

3. A class can only contain Attributes.

```
self.feature->forAll(f | f.oclIsKindOf(Attribute))
```

- GRC (stereotype of Class)

1. A GRC Class is active, that is an object of the Class maintains its own thread of control.

```
self.isActive
```

2. A GRC Class can only have PortAggregation Associations.

```
self.associations->forAll(a |
    a.stereotype.name = ''portaggregation'')
```

3. A GRC Class is linked to at least one PortType Class through a PortAggregation Association.

```
self.associations.size >= 1
```

4. The target of an Attribute of a GRC Class is an Instance.

```
self.allAttributes->forAll(af | af.targetScope = #instance)
```

5. The multiplicity of an Attribute of a GRC Class is 1.

```
self.allAttributes->forAll(af | af.multiplicity.max = 1)
```

6. The visibility of an Attribute of a GRC Class is private.

```
self.allAttributes->forAll(af | af.visibility = #private)
```

7. The value of an Attribute of a GRC Class can be modified.

```
self.allAttributes->forAll(af | af.changeability = #none)
```

8. The type of an Attribute of a GRC Class is either a PortType Class, a DataType, or a Binding of a template DataType.

```
self.allAttributes->forAll(af |
    af.type.oclIsKindOf(Class) and
        af.type.stereotype.name = ''porttype'' or
    af.type.oclIsKindOf(DataType) or
    af.type.oclIsKindOf(Binding))
```

9. If the type of an Attribute of a GRC Class is a PortType Class, then there is a PortAggregation Association between the GRC Class and the PortType Class.

```
self.allAttributes->forAll(af |
    af.type.oclIsKindOf(Class) and
        af.type.stereotype.name = ''porttype'' implies
            self.allOppositeAssociationEnds->exists(ae |
                ae.type = af.type))
```

10. If the type of an Attribute of a GRC Class is a Binding of a template DataType, then every argument of the Binding is either a DataType or a PortType Class.

```
self.allAttributes->forAll(af |
    af.type.oclIsKindOf(Binding) implies
        af.type.arguments->forall(me |
            me.oclIsKindOf(Class) and
                me.stereotype.name = ''porttype'' or
            me.oclIsKindOf(DataType)))
```

11. If the type of an Attribute of a GRC Class is a Binding of a template DataType, and an argument of the Binding is a PortType Class, then there is a PortAggregation Association between the GRC Class and the PortType Class.

```
self.allAttributes->forAll(af |
    af.type.oclIsKindOf(Binding) and
        af.type.arguments->forall(me |
            me.oclIsKindOf(Class) and
                me.stereotype.name = ''porttype'' implies
                    self.allOppositeAssociationEnds->exists(ae |
                        ae.type = me)))
```

- PortType (stereotype of Class)

1. A PortType Class is not active.

```
not self.isActive
```

2. A PortType Class is linked to exactly one GRC Class through a PortAggregation Association.

```
self.associations->select(a |
    a.stereotype.name = ''portaggregation'').size = 1
```

3. A PortType Class is linked to at least one other PortType Class through a PortLink Association.

```
self.associations->select(a |
    a.stereotype.name = ''portlink'').size >= 1
```

4. A PortType Class has one Attribute.

```
self.allAttributes.size = 1
```

5. The target of an Attribute of a PortType Class is the Classifier itself.

```
self.allAttributes->forAll(af | af.targetScope = #classifier)
```

6. The multiplicity of an Attribute of a PortType Class is 1.

```
self.allAttributes->forAll(af | af.multiplicity.max = 1)
```

7. The visibility of an Attribute of a PortType Class is public.

```
self.allAttributes->forAll(af | af.visibility = #public)
```

8. The initial value of the Attribute of a PortType Class cannot be modified. The identifier of the Attribute is *events*, and the type of the Attribute is the Classifier *Set*.

```
self.allAttributes->forAll(af |
    af.changeability = #frozen and
        af.name = ''events'' and af.type.name = ''Set'')
```

Subpackage *Core - Relationships*

- Association

1. An Association has either the stereotype <<PortAggregation>> or the stereotype <<PortLink>>.

```
self.stereotype.name = ''portaggregation'' or
    self.stereotype.name = ''portlink''
```

2. An Association has exactly two AssociationEnds.

```
self.connections.size = 2
```

3. The AssociationEnds of an Association are navigable.

```
self.connections->forAll(ae | ae.isNavigable)
```

4. The AssociationEnds of an Association are not ordered,

```
self.connections->forAll(ae | ae.ordering = #unordered)
```

5. The target of an AssociationEnd of an Association is an Instance.

```
self.connections->forAll(ae | ae.targetScope = #instance)
```

6. The multiplicity of an AssociationEnd of an Association is 1.

```
self.connections->forAll(ae | ae.multiplicity.max = 1)
```

7. The instance of an Association cannot be modified.

```
self.connections->forAll(ae | ae.changeability = #frozen)
```

8. The visibility of an Association is public.

```
self.connections->forAll(ae | ae.visibility = #public)
```

- PortAggregation (stereotype of Association)

1. In a PortAggregation Association, the Classifier of one AssociationEnd is a GRC Class and the Classifier of the other AssociationEnd is a PortType Class. The GRC Class is a composition aggregation of the PortType Class.

```
self.stereotype.name = ''portaggregation'' implies
    self.connections->exists(ae1, ae2 |
        ae1 <> ae2 and
            ae1.type.stereotype.name = ''grc'' and
                ae1.type.stereotype.name = ''porttype'')
```

- PortLink (stereotype of Association)

1. In a PortLink Association, the Classifier of each AssociationEnd is a distinct PortType Class. There is no aggregation relationship between the two PortType Classes.

```
self.stereotype.name = ''portlink'' implies
    self.connections->exists(ae1, ae2 |
        ae1 <> ae2 and
            ae1.type.stereotype.name = ''porttype'' and
                ae1.type.stereotype.name = ''porttype'')
```

UML Package ***BehavioralElements***

Subpackage ***Collaborations***

- Collaboration

1. A Collaboration may only contain ClassifierRoles and AssociationRoles.

```
self.ownedElements->forAll(r |
    r.oclIsKindOf(ClassifierRole) or
        r.oclIsKindOf(AssociationRole))
```

2. If the Classifier of a ClassifierRole is a GRC Class, then there is at least one AssociationRole whose Association is a PortAggregation Association, and the type of one of the AssociationEndRoles of the AssociationRole is the ClassifierRole.

```
self.ownedElements->forAll(cr |
    cr.oclIsKindOf(ClassifierRole) and
        cr.base.stereotype.name = ''grc'' implies
           self.ownedElements->exists(ar |
              ar.oclIsKindOf(AssociationRole) and
                 ar.base.stereotype.name = ''portaggregation''
                    and ar.connections->exists(aer |
                       aer.type = cr)))
```

3. If the Classifier of a ClassifierRole cr_1 is a GRC Class, then there is at least one AssociationRole ar_1 whose Association is a PortAggregation Association, and whose AssociationEndRoles are aer_1 and aer_2, such that (i) the type of aer_1 is the ClassifierRole cr_1; (ii) the type of aer_2 is another ClassifierRole cr_2; and (iii) there exists one AssociationRole ar_2 whose Association is a PortLink Association, and the type of one of the AssociationEndRoles of the AssociationRole ar_2 is the ClassifierRole cr_2.

```
self.ownedElements->forAll(cr1 |
  cr1.oclIsKindOf(ClassifierRole) and
    cr1.base.stereotype.name = ''grc'' implies
      self.ownedElements->exists(ar1 |
        ar1.oclIsKindOf(AssociationRole) and
          ar1.base.stereotype.name = ''portaggregation''
           and ar1.connections->exists(aer1, aer2 |
             aer1.type = cr1 and
               self.ownedElements->forAll(cr2 |
                 cr2.oclIsKindOf(ClassifierRole) and
                   aer2.type = cr2 and
                     self.ownedElements->exists(ar2 |
                       ar2.oclIsKindOf(AssociationRole) and
                         ar2.base.stereotype.name = ''portlink''
                          and ar2.connections->exists(aer3 |
                            aer3.type = cr2))))))
```

4. If the Classifier of a ClassifierRole is a PortType Class, then there is exactly one AssociationRole whose Association is a PortAggregation Association, and the type of one of the AssociationEndRoles of the AssociationRole is the ClassifierRole.

```
self.ownedElements->forAll(cr |
    cr.oclIsKindOf(ClassifierRole) and
        cr.base.stereotype.name = ''porttype'' implies
           self.ownedElements->select(ar |
              ar.oclIsKindOf(AssociationRole) and
                 ar.base.stereotype.name = ''portaggregation''
```

```
and ar.connections->exists(aer |
  aer.type = cr).size = 1))
```

5. If the Classifier of a ClassifierRole is a PortType Class, then there is at most one AssociationRole whose Association is a PortLink Association, and the type of one of the AssociationEndRoles of the AssociationRole is the ClassifierRole.

```
self.ownedElements->forAll(cr |
    cr.oclIsKindOf(ClassifierRole) and
        cr.base.stereotype.name = ''porttype'' implies
            self.ownedElements->select(ar |
                ar.oclIsKindOf(AssociationRole) and
                    ar.base.stereotype.name = ''portlink''
                        and ar.connections->exists(aer |
                            aer.type = cr).size <= 1))
```

- ClassifierRole

1. The multiplicity of a ClassifierRole is 1.

```
self.multiplicity.max = 1
```

2. The Classifier of a ClassifierRole is either a GRC Class or a PortType Class.

```
self.base.stereotype.name = ''grc'' or
    self.base.stereotype.name = ''porttype''
```

- AssociationRole

1. The multiplicity of an AssociationRole is 1.

```
self.multiplicity.max = 1
```

2. The Association of an AssociationRole is either a PortAggregation Association or a PortLink Association.

```
self.base.stereotype.name = ''portaggregation'' or
    ar.base.stereotype.name = ''portlink''
```

3. An AssociationRole has exactly two AssociationEndRoles.

```
self.connections.size = 2
```

4. If the Association of an AssociationRole is a PortAggregation Association, then the Classifier of the type of one AssociationEndRole is a GRC Class and the Classifier of the type of the other AssociationEndRole is a PortType Class. The PortAggregation Association links the GRC Class and the PortType Class.

```
self.base.stereotype.name = ''portaggregation'' implies
    and self.connections->exists(aer1, aer2 |
```

```
aer1.type.base.stereotype.name = ''grc'' and
    aer2.type.base.stereotype.name = ''porttype'' and
        self.base.connections->exists(ae1, ae2 |
            ae1.type = aer1.type.base and
                ae1.type = aer2.type.base))
```

5. If the Association of an AssociationRole is a PortLink Association, then the Classifier of the type of each AssociationEndRole is a distinct PortType Class. The PortLink Association links the two PortType Classes.

```
self.base.stereotype.name = ''portlink'' implies
    and self.connections->exists(aer1, aer2 |
        aer1.type.base.stereotype.name = ''porttype'' and
            aer2.type.base.stereotype.name = ''porttype'' and
                self.base.connections->exists(ae1, ae2 |
                    ae1.type = aer1.type.base and
                        ae1.type = aer2.type.base))
```

- Interaction

1. The sender of every Message in an Interaction is a ClassifierRole whose Classifier is a GRC Class. The ClassifierRole is in the context of the Interaction.

```
self.messages->forAll(m |
    m.sender.base.stereotype.name = ''grc'' and
        self.context->includes(m.sender))
```

2. The receiver of every Message in an Interaction is a ClassifierRole whose Classifier is a GRC Class. The ClassifierRole is in the context of the Interaction.

```
self.messages->forAll(m |
    m.receiver.base.stereotype.name = ''grc'' and
        self.context->includes(m.receiver))
```

3. The communication connection of every Message in an Interaction is an AssociationRole whose Association is a PortLink Association. The PortLink Association links two PortType Classes that are part of two GRC classes. The two GRC Classes correspond to the Classifiers of the ClassifierRoles representing the sender and receiver of the Message.

```
self.messages->forAll(m |
 m.communicationConnection.base.stereotype.name = ''portlink'' and
  m.communicationConnection.base.connections->exists( ae1, ae2 |
   ae1.type.stereotype.name = ''porttype'' and
    ae2.type.stereotype.name = ''porttype'' and
     self.context.ownedElements->exists(ar1, ar2 |
```

```
ar1.connections->exists(aer1, aer2 |
  aer1.type.base = ae1.type and
    aer2.type.base = m.sender.base and
      ar2.connections->exists(aer3, aer4 |
        aer3.type.base = ae2.type and
          aer4.type.base = m.receiver.base)))))
```

Subpackage ***StateMachines***

- StateMachine

  1. The context of a StateMachine is a GRC Class.

     ```
     self.context.oclAsType(Class).stereotype.name = ''grc''
     ```

- Transition

  1. The source and target of a Transition is a State.

     ```
     self.source.oclIsKindOf(State) and
         self.target.oclIsKindOf(State)
     ```

  2. A Transition has one trigger Event.

     ```
     self.trigger.size = 1
     ```

  3. A Transition has one Guard.

     ```
     self.guard.size = 1
     ```

  4. A Transition has one effect Action.

     ```
     self.effect.size = 1
     ```

  5. The trigger Event of a Transition is a SignalEvent.

     ```
     self.trigger.oclIsKindOf(SignalEvent)
     ```

  6. The trigger Event of a Transition has no Parameter.

     ```
     self.trigger.parameters.isEmpty
     ```

  7. The effect Action of a Transition is synchronous.

     ```
     not self.effect.isAsynchronous
     ```

- State

  1. A State is either a SimpleState or a CompositeState.

     ```
     self.oclIsKindOf(SimpleState) or
         self.oclIsKindOf(CompositeState)
     ```

2. A State has no exit Action.

```
self.exit.isEmpty
```

3. A State has no activity Action.

```
self.doActivity.isEmpty
```

4. A State has no deferrable Events.

```
self.deferrableEvents.isEmpty
```

- CompositeState

1. A CompositeState is not concurrent.

```
self.oclIsKindOf(CompositeState) implies
    not self.oclAsType(CompositeState).isConcurrent
```

2. A subvertex of a CompositeState is a State.

```
self.oclIsKindOf(CompositeState) implies
    self.oclAsType(CompositeState).subvertices->forAll(sv |
        sv.oclIsKindOf(State))
```

4.4 Case Study – A Distributed Navigation Controller

We illustrate the UML-based modeling technique for real-time reactive systems using a simplified version of a road traffic control system. The system ensures collision-free coordinated motion of vehicles traveling through a junction of two perpendicular roads. We focus on the design of controllers that interact with vehicles through sensors and actuators to determine the proximity of vehicles at the intersection and to channel them through the crossing. The following features characterize the requirements for the control system.

- At the proximity of the intersection, each road is divided into six lanes; there are three lanes for *incoming* traffic in each of the northbound, southbound, eastbound, and westbound directions.
- In every direction, vehicles in the right lane turn right, vehicles in the middle lane go straight, and vehicles in the left lane turn left.
- Vehicles approaching in a lane enter the crossing on a first-in-first-out basis.
- Vehicles cross the intersection in a finite amount of time; no vehicle stops in the intersection.
- Incoming vehicles in the four *right* lanes are allowed inside the crossing independent of any other lane; they are collision-free. However, traffic lights regulate all lanes.
- Incoming vehicles in the *middle* and *left* lanes need to wait until they are granted permission to enter the crossing.

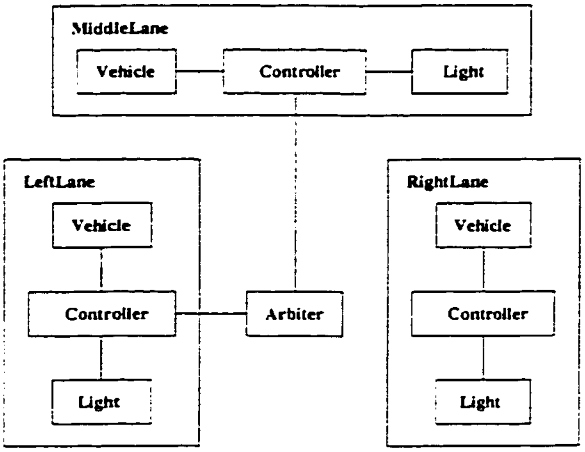


Figure 7: Architecture for Road Traffic Control System.

- Vehicles in at most two middle or left lanes can be granted access simultaneously, only if the two lanes are collision-free.
- When an approaching vehicle is detected in a middle or a left lane ml_1, the controller for the lane requests for access rights to the crossing from the arbiter.
  - If the intersection is empty, or only right lane vehicles are inside, then vehicles in lane ml_1 gain access immediately.
  - If the intersection is already opened to one middle or left lane ml_2, then vehicles in lane ml_1 gain access only if lanes ml_1 and ml_2 are collision-free.
  - If the intersection is already opened to two middle or left lanes ml_2 and ml_3, then vehicles in lane ml_1 need to wait until one of the lanes relinquishes access rights to the crossing. If lane ml_2 returns access rights, then lane ml_1 must be compatible with lane ml_3 in order to be granted access. Otherwise, vehicles in lane ml_1 must wait for lane ml_3 to give up its access rights.
- When a lane requests for access rights, the lane queues up for allocation. Access is granted to lanes in the order in which they request for permission.
- When a lane obtains access rights, it must surrender these rights within a certain time interval.

The intersection is a *shared resource* that is allocated to vehicles in such a way that the following properties are satisfied.

- *Liveness*: every vehicle at the intersection obtains the resource within a finite amount of time; there is neither deadlock nor starvation.
- *Safety*: vehicles do not collide while crossing the intersection.

The control algorithm is applicable to a wide range of situations including the coordination of collision-free motion of autonomous robots in a factory floor divided into mutually perpendicular workspaces.

4.4.1 Modeling

In each of the twelve incoming lanes, a sensor detects vehicles approaching the intersection, a traffic light regulates the flow of vehicles, and a controller monitors the traffic. An arbiter allocates access rights to the controllers as needed. The vehicles and the traffic lights are environmental entities; the controllers and the arbiter are system entities. For each lane, a subsystem models the communication configuration for the vehicles, the traffic light, and the controller. The overall reactive unit consists of twelve such subsystems, and one arbiter interacting with the middle and left lane controllers. Figure 7 shows the overall architecture for the traffic control system. The figure shows the arbiter and subsystems for the three incoming lanes.

All vehicles approach and leave the intersection subject to the same temporal behavior. On approaching, each vehicle informs the respective controller through the sensor, and either receives an immediate go-ahead to traverse the crossing or waits for clearance. The waiting time for a vehicle depends on the status of the lights, the number of vehicles in front of it in the lane, and the number of controllers that are waiting for resource allocation. Vehicles in the four right lanes cross the intersection simultaneously and independent of vehicles in other lanes.

When there is no vehicle approaching the intersection along a lane, the traffic light at that lane remains *red*. The controller for the lane switches the light from red to *green* when it holds access rights and there is an approaching vehicle in the lane. The controller switches the light from green to *yellow* when the allocation time for the lane is about to expire, or when all incoming vehicles in the lane have crossed the intersection, whichever occurs earlier; the light turns *red* when the time expires. The timing constraints on the behavior of the traffic lights are the same for the three kinds of lanes.

The role of the controller is to detect approaching vehicles through the sensor, and to manage the status of the traffic light. The timing constraints for controllers monitoring the flow of traffic approaching the intersection in the middle and left lanes are different from those for the right lanes. Right lane controllers have no interaction with the arbiter. Controllers for middle and left lanes request for access rights to the intersection from the arbiter whenever there is an approaching vehicle in the lane. Upon receiving the resource, the controller turns the light green, and gives the go-ahead to vehicles in the lane during the allocated time. The time-dependent interaction between the controllers and the lights is subject to the flow of traffic. For instance, the controller may switch the light from red to yellow before the end of the time-out period if there is no other vehicle approaching in the lane.

The arbiter optimizes resource utilization with concurrent allocation, ensures collision-free access, and prevents starvation. It processes requests for the resource from controllers for the middle and left lanes. It allocates the resource to at most two lanes concurrently, and on a first-in-first-out basis. The resource is to be released by the lane within a certain time interval. The arbiter maintains a queue of identifiers for requests from the lanes. It uses a table for determining the compatibility of lanes accessing the resource concurrently. The table enumerates the twelve scenarios for concurrent collision-free crossing of the intersection by vehicles in two different lanes. The main role of the arbiter is to prevent collision situations in the intersection and to allocate access rights with fairness and according to certain timing constraints.

4.4.2 Abstract Behavior

The time-dependent behavior of the system conforms to the following rules.

- Traffic lights:
  - A light remains red as long as there is no vehicle approaching or waiting at the crossing.
  - The controller switches the light from red to green, and subsequently from green to yellow.
  - The light switches to red within a period of 3 to 4 time units after turning yellow.
- Vehicles:
  - A vehicle sends a message to the respective controller on approaching the intersection.
  - The controller subsequently sends a go-ahead message to the vehicle when it holds access rights to the crossing.
  - The vehicle enters the crossing within 2 time units of receiving the go-ahead message.
  - The vehicle leaves the crossing within 5 time units of receiving the go-ahead message.
- Right lane controllers:
  - A right lane controller remains idle as long as there is no vehicle approaching or waiting at the crossing.
  - The controller switches the light from red to green within 2 time units of receiving a message from the first vehicle approaching the intersection.
  - The controller sends a go-ahead message to the first vehicle at the intersection within 1 to 2 time units of switching the light to green.
  - The controller receives messages from other approaching vehicles while the light is green.
  - The controller records the identifiers of the approaching vehicles in a queue as they are received.
  - The controller authorizes vehicles to cross the intersection during the allocation period of 10 to 40 time units, and removes the identifiers of the processed vehicles from the queue.
  - The controller switches the light from green to yellow within 1 time unit of being timed out.
  - If there is no vehicle in the queue, the controller goes back to idle; otherwise, it reactivates and switches the light from red to green within 20 to 30 time units of switching the light to yellow. This time gap may be used to allow for pedestrian crossing in an extended version of the system.
- Middle and left lane controllers:
  - A middle or left lane controller remains idle as long as there is no vehicle approaching or waiting at the crossing.
  - The controller requests for access rights from the arbiter within 2 time units of receiving a message from the first vehicle approaching the intersection.

- Upon receiving permission from the arbiter, the controller switches the light from red to green within 2 time units.
- The controller sends a go-ahead message to the first vehicle at the intersection within 1 to 2 time units of switching the light to green.
- The controller receives messages from other approaching vehicles while the light is green.
- The controller records the identifiers of the approaching vehicles in a queue as they are received.
- The controller authorizes vehicles to cross the intersection during the allocation period of 10 to 40 time units, and removes the identifiers of the processed vehicles from the queue.
- The controller switches the light from green to yellow within 1 time unit of being timed out.
- The controller returns the resource to the arbiter within 6 to 8 time units of turning the light from green to yellow.
- If there is no vehicle in the queue, the controller goes back to idle; otherwise, it reactivates and sends a new request for access rights to the arbiter within 20 to 30 time units of returning the resource. This time gap allows for other lanes to be processed without starvation.

- Arbiter:
  - The arbiter remains idle as long as there is no request for access rights from middle and left lane controllers.
  - When it receives a first request, it allocates the resource immediately.
  - If the resource is returned before any other request is received, the arbiter goes back to idle.
  - The arbiter records requests for access rights in the queue as they are received.
  - If the arbiter receives another request before the first controller returns the resource, the arbiter either
    * allocates the resource concurrently to the second controller if the two are compatible, or
    * waits for the first one to return the resource if the two are not compatible.
  - If one controller holds the resource, and there are requests in the queue, then when the controller returns the resource, the arbiter allocates the resource to the first one in the queue, and allocates the resource to the second one concurrently if the two are compatible. Otherwise, it waits for the first one to return the resource.
  - If two controllers hold the resource concurrently, and there is no request in the queue, then when one of the controllers returns the resource, the arbiter goes into the state where only one controller is holding the resource and there is no pending request.
  - If two controllers hold the resource concurrently, and there are requests in the queue, then when one of the controllers returns the resource, the arbiter allocates the resource to the first one in the queue if it is compatible with the controller still holding the resource. Otherwise, it waits for this controller also to return the resource.

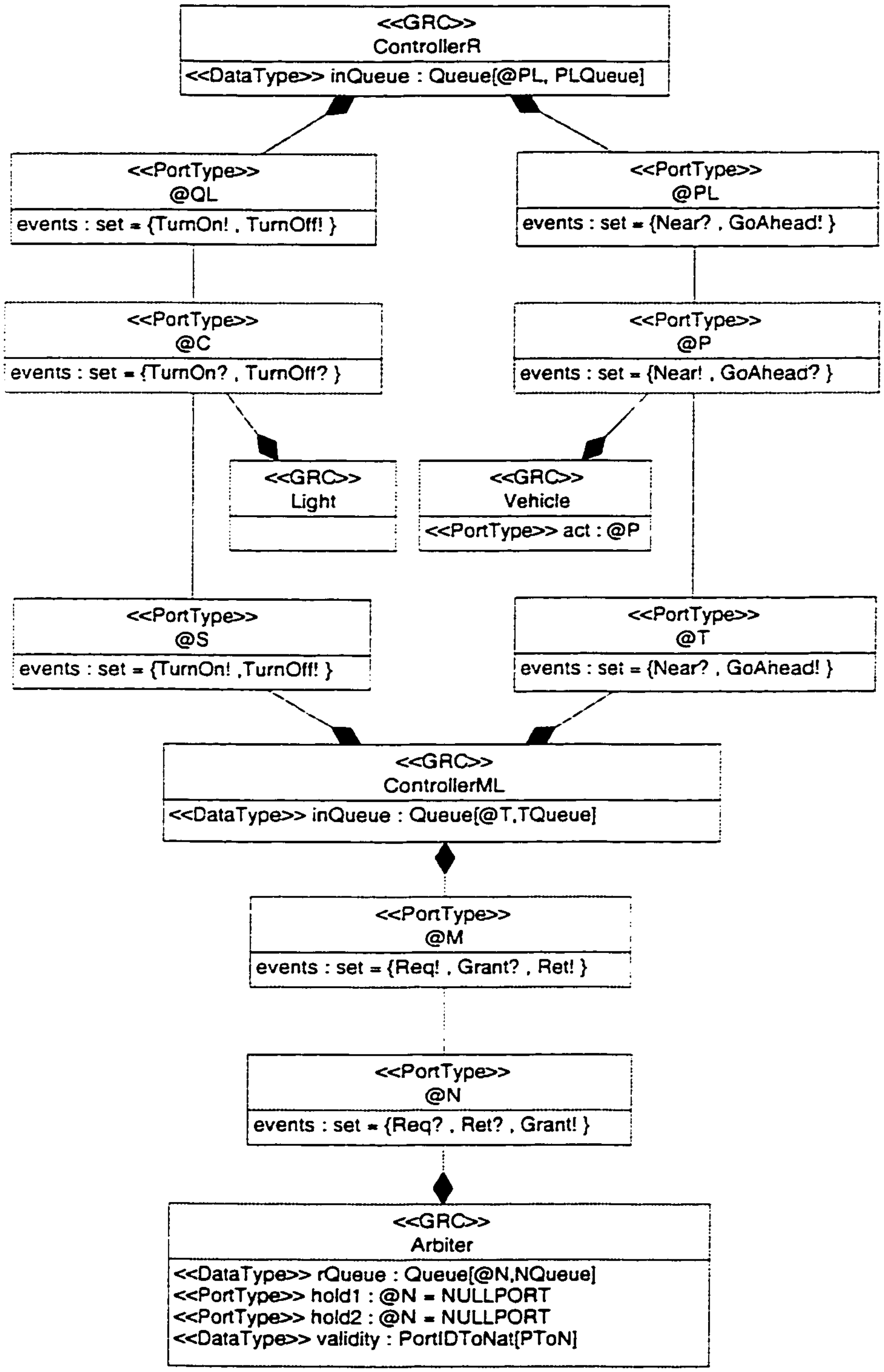


Figure 8: Class Diagram for Road Traffic Control System.

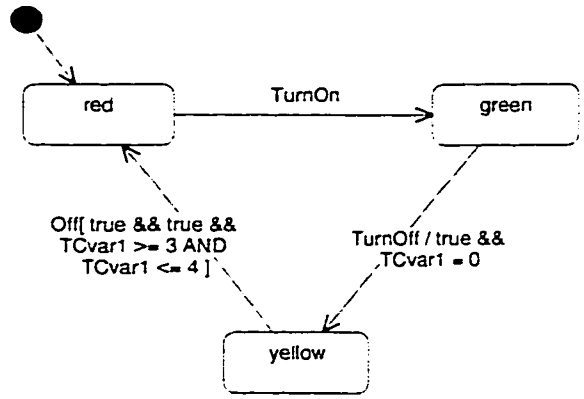


Figure 9: Statechart Diagram for GRC Light.

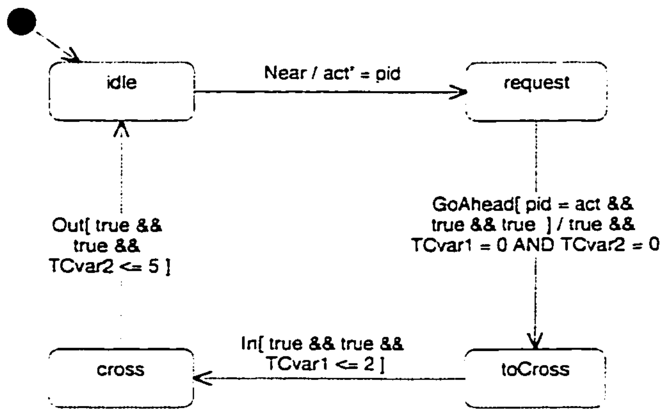


Figure 10: Statechart Diagram for GRC Vehicle.

4.4.3 UML Model

We model the vehicles, traffic lights, controllers and arbiter as generic reactive classes describing the structure and behavior of the models. Incorporating instances of the reactive classes in subsystems, we configure object collaborations for each of the twelve lanes, with links between ports of the controller, light and vehicle objects for communication. The overall system includes the twelve subsystems and an instance of the class specifying the arbiter. Port links between the arbiter and the controllers model the communication mechanism between these objects.

The *Light* and *Vehicle* generic reactive classes (GRC's) model the traffic lights and vehicles respectively. The class *Light* includes port type C for interaction with the controller. The class *Vehicle* includes port type P to communicate with the controller. A vehicle can include several instances of the port type for communication with the different controllers, depending on its direction. The *Vehicle* class includes the attribute *act*, a variable of the port type P, to identify the port of communication with the controller. The GRC *Arbiter* includes port type N for interaction with controllers. The class includes an instance of the abstract data type *Queue* to keep track of requests from the controllers, an instance of the compatibility table *PortIDToNat* for determining compatible lanes, and two variables of the port type M as attributes identifying the controllers that are holding the resource concurrently.

The GRC *ControllerR* models controllers regulating traffic in right lanes. The class aggregates two port types *PL* and *QL* for controller interaction with the vehicles and the traffic light respectively. The GRC *ControllerML* models controllers regulating traffic in middle and left lanes. The class aggregates three port types, *S*, *T* and *M*, for communication with the traffic light, vehicle and arbiter objects, respectively. Each controller has an instance of the abstract data type *Queue*, to keep track of vehicles approaching and leaving the intersection. Subsystems for the right lanes include an instance of GRC *ControllerR*; subsystems for the middle and left lanes include an instance of GRC *ControllerML*. The two controller classes are structurally homomorphic, but behaviorally different. *ControllerR* and *ControllerML* objects provide similar functionalities, but subject to different timing constraints.

A controller switches the light from red to green with the message *TurnOn*, and switches the light from green to yellow with the message *TurnOff*. The controller and the light synchronize on the event *TurnOn* entering the states *busy* and *green* respectively. The event *TurnOff* causes the light to enter the state *green* and the controller to enter the state *busy* simultaneously. These events occur at a port of type *C* of the light and at a port of type *QL* of a right lane controller or at a port of type *S* of a middle or left lane controller.

The event *Near* occurs at a port of type *P* of a vehicle, and simultaneously at a port of type *PL* of a right controller or at a port of type *T* of a middle or left controller. These occurrences correspond to messages that vehicles send to the controller on approaching the intersection; they cause simultaneous transitions in the statechart diagrams of the vehicle and controller objects. The synchronized message *Near* causes the vehicle to go from the state *idle* to the state *request*, and the controller to enter the state *activate* simultaneously. The controller also accepts the *Near* message in the states *activate*, *busy*, and *monitor*. The controller sends the message *GoAhead* to the vehicle through the same port link to allow it to enter the intersection.

The middle or left lane controller sends the message *Req* to the arbiter and waits for its response modeled by the message *Grant*. A controller returns the resource to the arbiter with the message *Ret*. The arbiter sends the message *Grant* to the controller to allocate it the resource. The controllers and the arbiter synchronize on the events *Req*, *Grant*, and *Ret*. The events occur at a port of type *M* of the controller and a port of type *N* of the arbiter simultaneously.

Figure 8 shows the class diagram for the GRC and PortType classes describing the structure of the system. It includes binary associations of the stereotype PortLink between the PortType classes to indicate the compatible port types. The PortType classes specify the set of events allowed through ports of these types. The statechart diagrams shown in Figures 9, 10, 11, 12 and 13 model the reactive behavior of the traffic lights, vehicles, controllers, and arbiter. The class diagram and the statechart diagrams together describe the reactive models of the environmental and system entities in the traffic controller system.

The collaboration diagram in Figure 14 shows the configuration of the subsystem for the right lane in the northbound direction. It includes one instance of the *ControllerR* and the *Light* classes, and three instances of the *Vehicle* classes. The configuration describes the synchronous communication mechanism between the controller and the light, and between the controller and the vehicles. Figure 15 shows the subsystem for the left lane in the northbound direction. The configuration includes a port of type *M* in the controller for interaction with the arbiter. Figure 16 shows the configuration of a traffic control system with controllers for the twelve lanes, and one arbiter interacting with controllers for middle and left lanes. There is one port link between the arbiter and each of the middle and left lane controllers. The behavior of the system can be

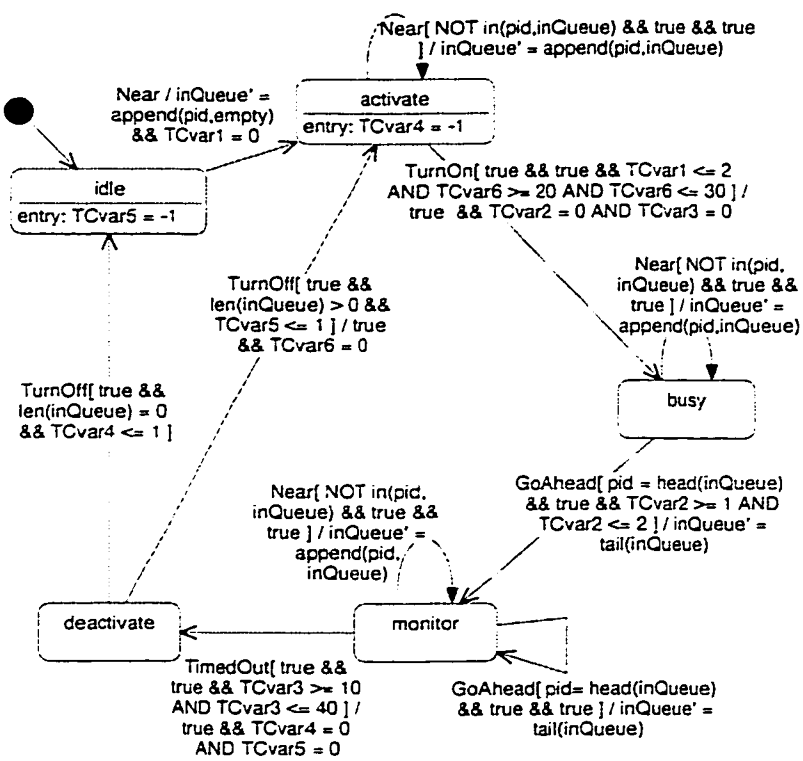


Figure 11: Statechart Diagram for Right Lane Controller GRC.

inferred by following the messages through the ports of the reactive objects in the collaboration diagram, and referring to the corresponding statechart diagrams.

The sequence diagram in Figure 17 shows a generic scenario for the subsystem describing objects at the right lane in the northbound direction. It shows that the controller allows vehicles inside the crossing without requesting permission from the arbiter. Figure 18 shows a sequence diagram for the northbound left lane subsystem. It shows that the controller requests for access rights from the arbiter and that vehicle *V6* has to wait until the arbiter allocates the resource. Figure 19 shows a sequence diagram describing a scenario involving the arbiter and objects from the subsystems for the left lane in the northbound direction and the middle lane in the southbound direction. As these two lanes are not compatible, the arbiter does not allocate the resource to both concurrently. Hence, southbound vehicles in the left lane need to wait before entering the crossing. A synthesis of the timing constraints given in the statechart diagrams yields the generic scenarios shown in the sequence diagrams. In these scenarios, the logical variables indicate absolute times for event occurrences. The linear inequalities in the sequence diagrams provide a graphic representation of the timing constraints on reactions.

4.4.4 Analyzing the Model

The design of the arbiter ensures safety, liveness, and dynamic resource allocation. By encapsulating timing constraints in the controller classes, we have aimed at promoting the arbiter to control nonuniform traffic flows. In our modeling technique, an instance of a generic reactive class has a single thread of control;

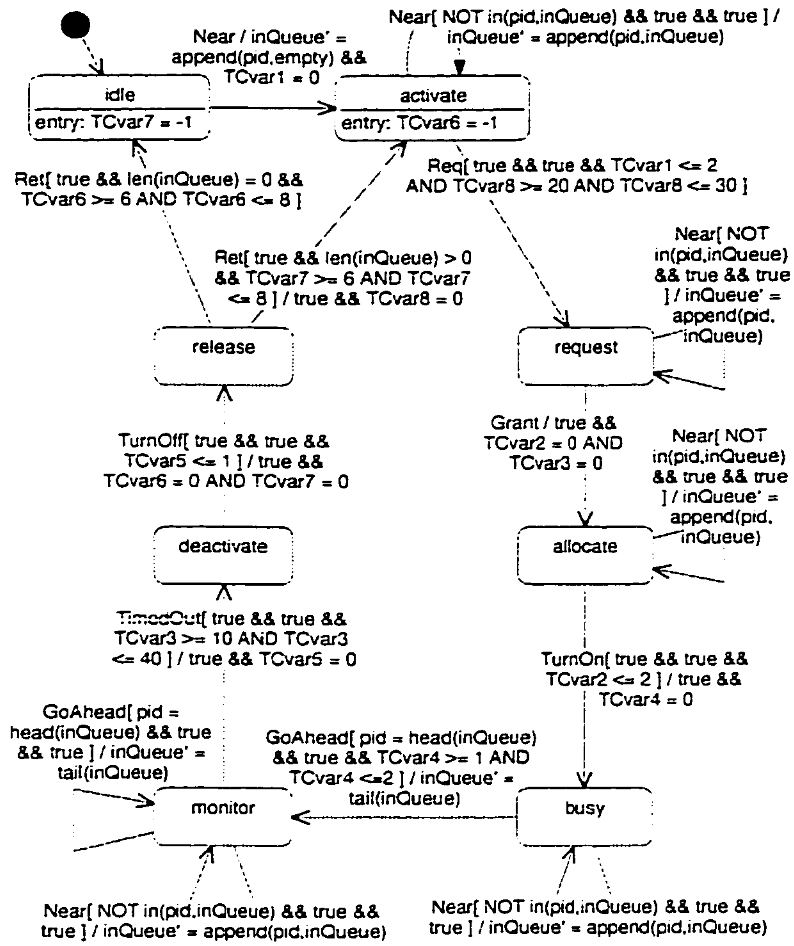


Figure 12: Statechart Diagram for Left/Middle Lane Controller GRC.

events can not occur concurrently within an object. However, concurrency in communicating reactive objects can be modeled. In our design, the arbiter allocates the resource to two controllers concurrently if they are compatible.

There are twelve patterns of traffic flow involving concurrent crossing of the intersection by vehicles in two middle or left lanes without collision. Table 1 defines the compatible pairs of lanes. Table 2 describes the identifiers for the controllers of the middle and left lanes, the corresponding port identifier of the arbiter, and the relative index in the compatibility table. Figure 20 shows the LSL trait *Table* defining the compatibility of lanes. The included trait *IntCycle* is available in [GH93]. The compatible pairs of lanes as defined in Table 1 can be deduced from the axioms in the trait *Table*. The predicate *validEntry* identifies the entries that are *true* in the table. Figure 21 shows the LSL trait *PortIDToNat* that defines the mapping f of port identifiers of the arbiter to indices of the table, and the predicate *isValid* that takes two port identifiers of the arbiter for determining the compatibility of lanes.

The resource allocation scheme prevents collision inside the intersection. When in the *idle* state, the arbiter does not grant the resource. When the arbiter is in one of the states *oneBusy*, *twoBusy* or *busyWait*, resources can be released by controllers, but the arbiter does not allocate the resource. The arbiter allocates

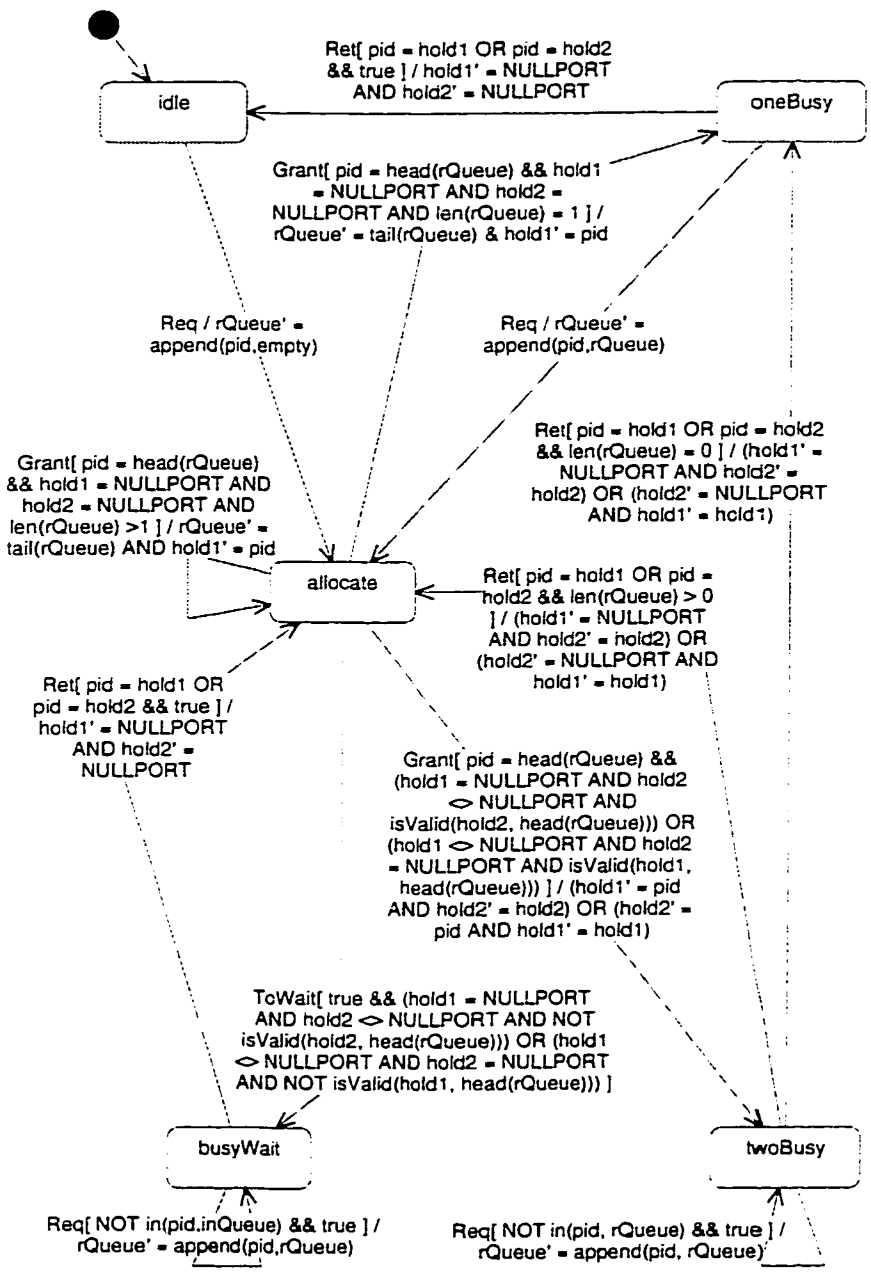


Figure 13: Statechart Diagram for GRC Arbiter.

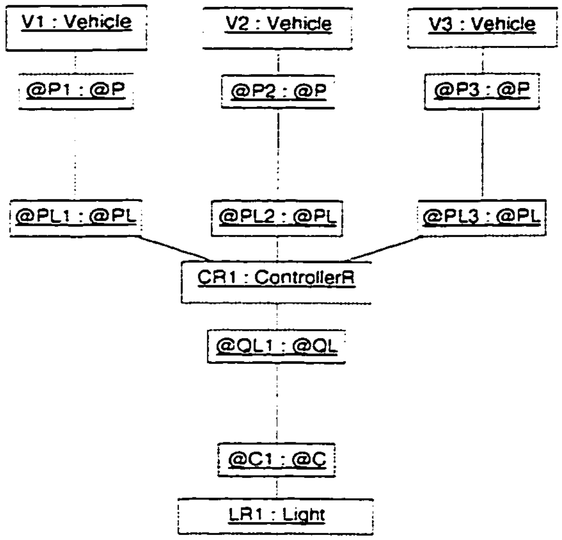


Figure 14: Collaboration Diagram for Northbound Right Lane.

the resource to a controller only when in the state *allocate*. The three resource allocation scenarios are as follows.

1. The arbiter allocates the resource and stays in state *allocate*. From transition specification R_6 of the reactive class, we infer that only one controller is accessing the resource, and there are pending requests in the queue.

2. The arbiter allocates the resource and goes into state *oneBusy*. From transition specification R_4, we infer that only one controller is accessing the resource, and there is no pending request.

3. The arbiter allocates the resource and goes into state *twoBusy*. From transition specification R_5, we infer that the two controllers utilizing the resource are compatible.

In each case, collision inside the intersection is precluded. The enabling conditions of transition specifications R_4, R_5 and R_6 are mutually exclusive, ensuring deterministic resource allocation.

When the arbiter has allocated the resource to one controller and assessed that the next controller in the queue is not compatible with the one already holding the resource, it goes into the state *busyWait*. The resource is not allocated to the controller at the front of the queue. Thus, when the arbiter is in state *busyWait*, vehicles in only one middle or left lane are crossing the intersection.

Safety and liveness

The design of the arbiter ensures the satisfaction of safety and liveness properties. Two middle and left lane controllers hold the resource concurrently only when the two lanes are compatible. This ensures that collision will never occur in the crossing. Liveness is guaranteed by the time-dependent behavior of controllers. They return the resource to the arbiter within a finite time period; the arbiter then allocates the resource to the lane waiting at the front of the queue.

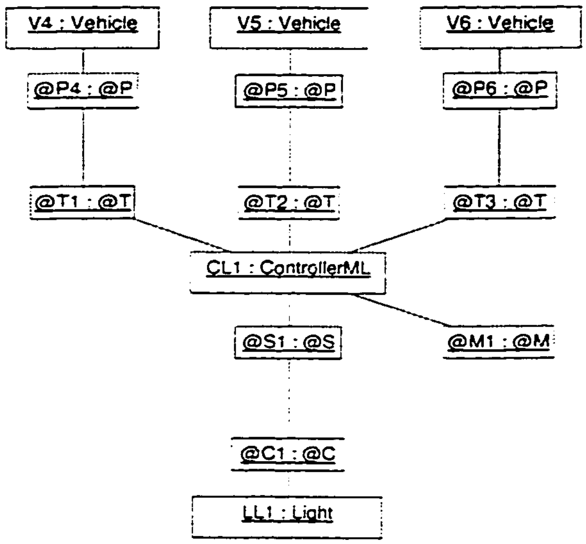


Figure 15: Collaboration Diagram for Northbound Left Lane.

Northbound middle	Southbound middle
Northbound middle	Northbound left
Northbound middle	Eastbound left
Southbound middle	Westbound left
Southbound middle	Southbound left
Southbound left	Northbound left
Eastbound middle	Westbound middle
Eastbound middle	Eastbound left
Eastbound middle	Southbound left
Westbound middle	Northbound left
Westbound middle	Westbound left
Westbound left	Eastbound left

Table 1: Compatible Traffic Lanes.

Dynamic resource allocation

Resource allocation is driven by the demand for access rights. When there is no traffic in a lane, the resource is not allocated to the respective controller; the traffic light in that lane remains red. Only lanes with a flow of traffic receive access rights for the resource. Resource allocation based solely on concurrency so as to optimize utilization and to minimize overall waiting time for vehicles may lead to starvation. A strict first-in-first-out allocation scheme rules out starvation and allows concurrency within compatibility rules. Consequently, the maximum waiting time for a controller request is $O(k)$, where k is the average number of requests in the queue. Preventing starvation for requests from controllers ensures there is no starvation for vehicles.

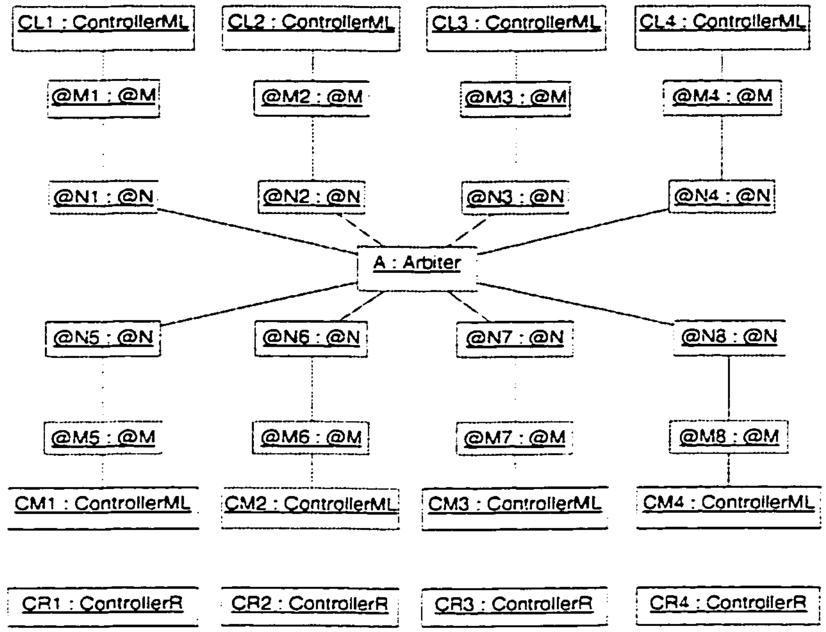


Figure 16: Collaboration Diagram for Road Traffic System.

Lane	Controller ID	Arbiter Port ID	Index
Northbound middle	CM1	N5	1
Northbound left	CL1	N1	2
Westbound middle	CM4	N8	3
Westbound left	CL4	N4	4
Southbound middle	CM2	N6	5
Southbound left	CL2	N2	6
Eastbound middle	CM3	N7	7
Eastbound left	CL3	N3	8

Table 2: Indices and Arbiter *port ids* for Traffic Lanes.

Controlling Nonuniform Traffic Flows

If the rates of traffic flow in different lanes vary substantially, then the controllers for managing the traffic lights will have different time constraints, but will maintain the same functionalities. In the object-oriented paradigm, the preservation of behavior ensures substitutability. However, this does not apply to the realm of embedded reactive systems. While preserving functional behavior, controllers regulating lanes with different rates of traffic flow resort to different time constraints. Since objects of one controller class cannot be substituted for different lanes, we need to specialize class derivations. We have studied three different forms of inheritance in [AAM96]. Of these, *behavioral inheritance* supports changes in timing constraints; that is, the minimal time delay can be increased or the maximal time delay can be decreased. Applying behavioral inheritance for the controller classes, we can specialize different controllers to regulate different patterns of traffic, while maintaining the same arbiter class. This illustrates an application of the benefits of object-orientation in our models.

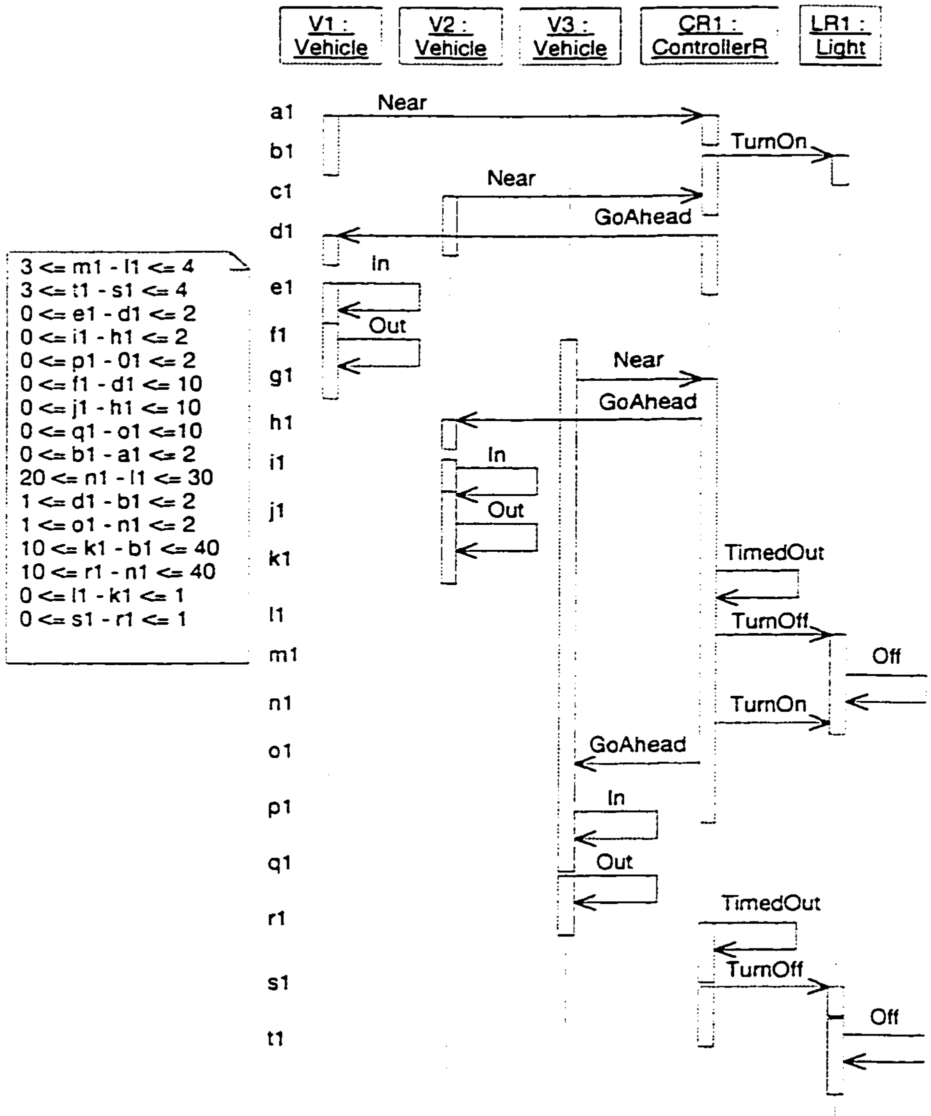


Figure 17: Sequence Diagram for Northbound Right Lane.

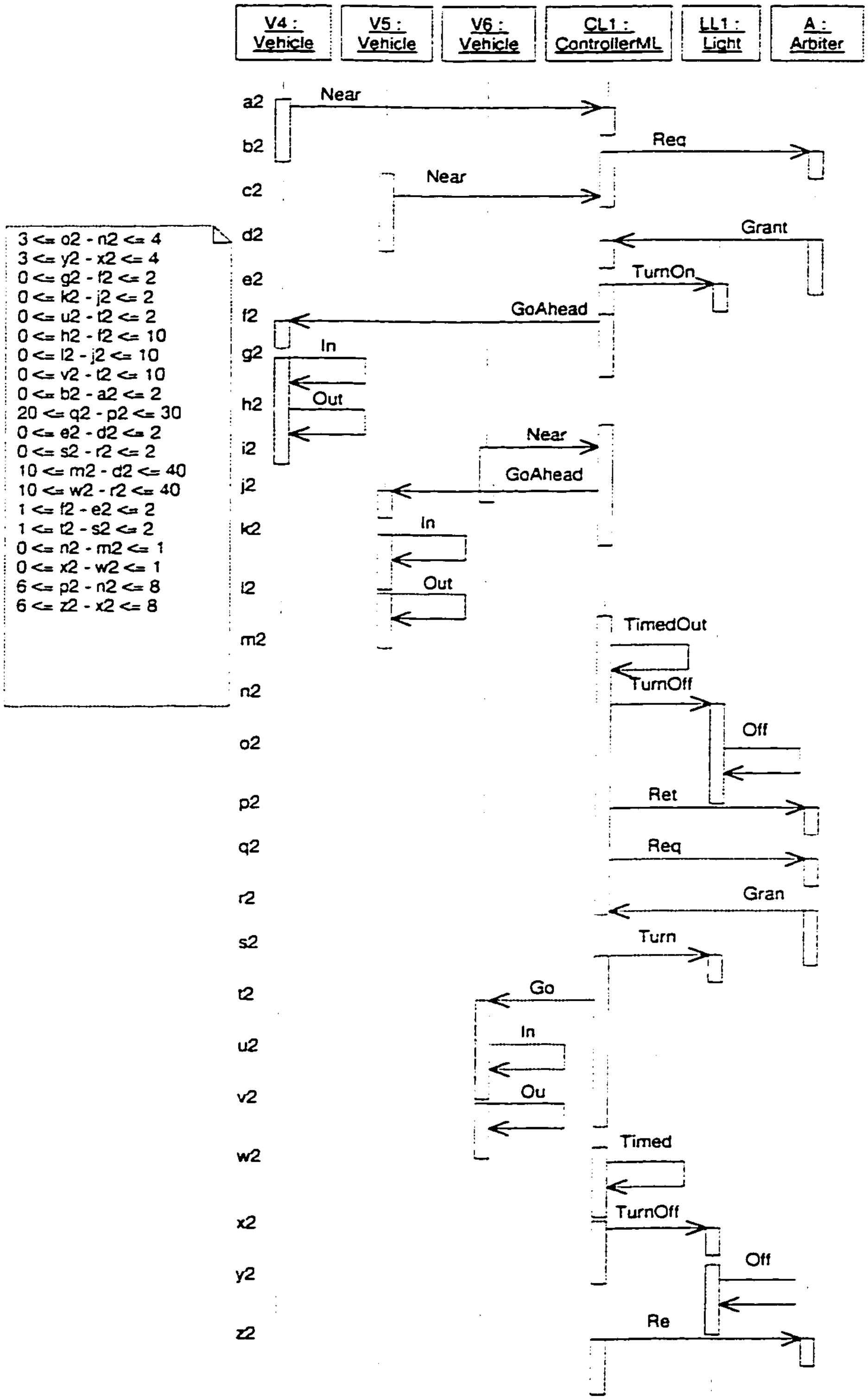


Figure 18: Sequence Diagram for Northbound Left Lane.

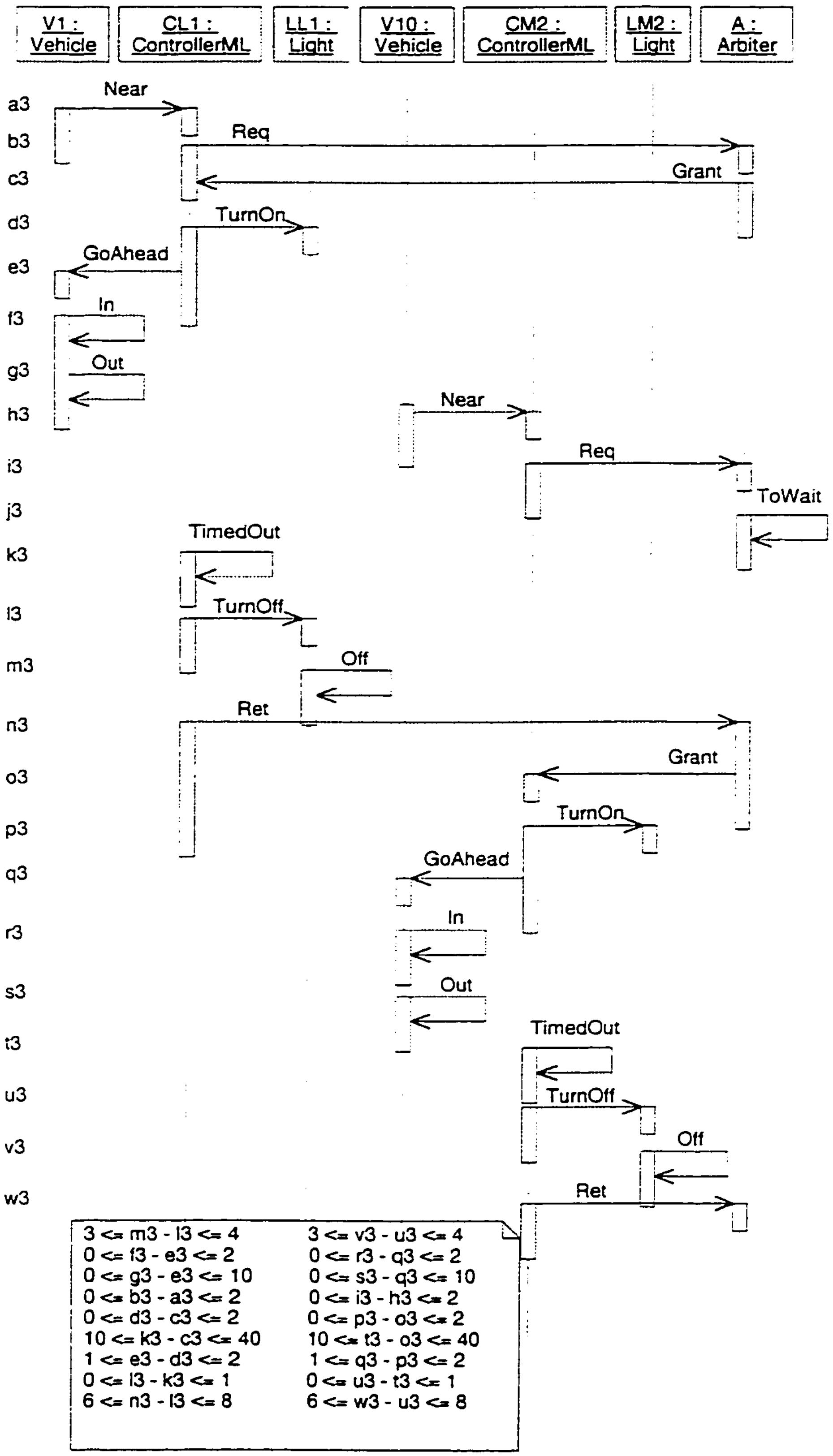


Figure 19: Sequence Diagram for Road Traffic System.

```
Table(T): trait
  includes IntCycle(0 for first, 7 for last, Index for N)
  introduces
    init: T → T
    __[__, __]: T, Index, Index → Bool
    validEntry: Index, Index → Bool
  asserts
    Table generated by init
    ∀ a : T, i, j : Index
      i ≥ 1 ∧ i ≤ 1 ⇒ init(a)[i, i+7] == true
      i ≥ 1 ∧ i ≤ 2 ⇒ init(a)[i, i+6] == false
      i ≥ 1 ∧ i ≤ 3 ⇒ init(a)[i, i+5] == false
      i ≥ 1 ∧ i ≤ 4 ⇒ init(a)[i, i+4] == true
      i ≥ 1 ∧ i ≤ 5 ⇒ init(a)[i, i+3] == false
      i ≥ 1 ∧ i ≤ 6 ⇒ init(a)[i, i+2] == false
      i ≥ 1 ∧ i ≤ 7 ⇒ init(a)[i, i+1] == true
      i ≥ 1 ∧ i ≤ 8 ⇒ init(a)[i, i] == false
      validEntry(i, j) == init(a)[i, j]
```

Figure 20: LSL specification for Trait Table.

```
PortIDToNat(PidToNat): trait
  includes Table
  introduces
    f: Port_Type → Index
    isValid: Port_Type, Port_Type → Bool
  asserts
    ∀ p1, p2 : Port_Type
      f(N5) == 1
      f(N1) == 2
      f(N8) == 3
      f(N4) == 4
      f(N6) == 5
      f(N2) == 6
      f(N7) == 7
      f(N3) == 8
      isValid(p1, p2) == validEntry(f(p1), f(p2))
```

Figure 21: LSL specification for Trait PortIDToNat.

4.5 Deriving a Formal Specification from a RTUML Model

We develop a mapping from the UML model of a reactive system onto formal specifications. A class diagram and its associated statechart diagram modeling a reactive entity succinctly map onto the formal description of a generic reactive class according to the template given in Figure 3. Features described in the class diagram, namely the *port types*, *attributes*, and *abstract data types* map onto the parametric port types, and the sections for Attributes, and Traits, respectively, in the textual description. Features specified in the statechart diagram map onto the sections for States, Events, Attribute-Function, Transition-Specifications, and Time-Constraints.

Each transition in the statechart diagram maps onto a transition specification in the textual description. The logical assertions for the port condition and the enabling condition are extracted from the *guard* of the transition; the assertion for the postcondition is extracted from the *action* of the transition. In defining the attribute function, the attributes that are modified in an *action* are considered to be *active* in the destination state of the transition. We use *logical clocks* in defining time constraints on reactions to transitions; a reactive object includes a set of logical clocks. The occurrence of a transition may cause the enabling of reactions in the form of internal or output events to occur at some point in a future time interval. On the enabling of a reaction, a corresponding logical clock is initialized to zero in the *action* of the transition causing the reaction. The lower and upper bounds for the time interval during which the reaction can occur are specified as logical assertions in the *guard* of the transition corresponding to the reaction. A negative value for a logical clock in the *action* of a transition indicates that the destination state is a *disabling* state for the reaction.

The UML collaboration diagram configuring a subsystem leads to a textual description based on the template shown in Figure 4. The reactive objects in the configuration map onto the Instantiate section, and the cardinality for each port type of the reactive object is derived from the port objects. The binary associations of the stereotype *PortLink* between ports of reactive objects lead to the port-links in the Configure section. A composition of collaboration diagrams defining a complex subsystem corresponds to the subsystems in the Include section.

This mapping is a one-to-one correspondence between elements of the UML model and those of the formal notation, and establishes a basis for automating the translation mechanism for extracting textual specifications from the UML diagrams. The formal notation allows (i) a self-contained specification of each reactive object model as a high-level class description, and (ii) checking the syntactic and semantic correctness of the specified models with an interpreter. Popistas [Pop99] describes a translator based on the mapping; the tool was implemented within Rational Rose environment. We use Rose to specify the UML model and implement the *Rose-GRC* translator using RoseScript. The script program gives access to classes, properties and methods, to update models and generate documentation in Rose. The translator operates within the Rose environment, and uses graphical interface constructs from Rose Extensibility Interface.

4.5.1 Formal Specification of the Road Traffic Control System

The Rose-GRC translator generates specifications consistent with the grammar of the specification language from well-formed RTUML class diagrams, statechart diagrams, and collaboration diagrams. We illustrate the translation process with the road traffic control system. Figures 22, 23, 24, 25 and 26 show the formal specifications generated by the translator from the RTUML model of the generic reactive entities described in the class diagram in Figure 8, and the statechart diagrams in Figures 9, 10, 11, 12 and 13, respectively. The LSL trait describing the data model *Queue* is available in [GH93]. Figures 27, 28 and 29 show the specifications for the configurations described in the collaboration diagrams in Figures 14, 15 and 16.

A subsystem configured with three vehicles, one controller and one traffic light has the following ports and communication links. Each vehicle has one port of type *P* for link to the controller. The controller has three ports of type *PL* for link to the vehicles. A right lane controller has one port of type *QL*, and a middle or left lane controller has one port of type *S*, for link to the traffic light it operates. The traffic light has one port of type *C* for link to the controller that operates it. A controller operates only one traffic light; the unique port of type *C* of the light links to the unique port of type *QL* of a right lane controller, or the unique port of type *S* of a middle or left lane controller. A middle or left lane controller has a port of type *M* for link to the arbiter. In the overall system, the arbiter has eight ports of type *N* for link to the middle and left lane controllers.

4.5.2 Validation and Verification

The main goal for rigorous analysis of design specifications is to detect flaws at an early stage in system development. An operational semantics for the abstract reactive model allows simulating the formal specification of a system design for validation against requirements. A logical semantics for the notation supports formal verification of desired system properties in the design, by demonstrating that the properties are logical consequences of the axiomatic description of the design.

Simulation of the formal model of a reactive system generates scenarios of the behavior of the system. Analyzing the data from the history of a simulation run provides valuable insight in the dynamic aspects of the model. Types of flaws that are detected through this mechanism include incorrect configurations, functionalities, and timing constraints. For instance, inconsistent timing constraints can lead to unsafe scenarios, and erroneous configurations can lead to deadlocks. A safety property can be verified to hold at each computation step in the simulation process.

While simulation provides confidence in the correctness of a design specification, a formal verification of certain properties is crucial in safety-critical applications. A verification methodology based on the abstract model provides a means for verifying safety and liveness properties in the formal specifications of a design. A translation mechanism generates axioms in the specification language of PVS from the formal specifications. Stating a property as a theorem, a user interacts with PVS theorem prover to determine whether the design satisfies the property. In particular, this technique is applicable to time-dependent properties whose satisfaction is determined by constraints on reactions to transitions. We have formulated a verification methodology based on the logical semantics of the abstract model, using PVS theorem prover.

```
Class Light [@C]
    Events: TurnOn?@C, TurnOff?@C, Off
    States: *red, green, yellow
    Attributes:
    Traits:
    Attribute-Function:
        red -> {}; green -> {}; yellow -> {};
    Transition-Specifications:
        R1: <red,green>; TurnOn(true); true => true;
        R2: <green,yellow>; TurnOff(true); true => true;
        R3: <yellow,red>; Off(true); true => true;
    Time-Constraints:
        TCvar1: R2, Off, [3, 4], {};
end
```

Figure 22: Formal specification for GRC Light.

```
Class Vehicle [@P]
    Events: Near!@P, GoAhead?@P, In, Out
    States: *idle, request, toCross, cross
    Attributes: act:@P
    Traits:
    Attribute-Function:
        idle -> {}; request -> {act};
        toCross -> {}; cross -> {};
    Transition-Specifications:
        R1: <idle,request>; Near(true); true => act' = pid;
        R2: <request,toCross>;
              GoAhead(pid = act); true => true;
        R3: <toCross,cross>; In(true); true => true;
        R4: <cross,idle>; Out(true); true => true;
    Time-Constraints:
        TCvar1: R2, In, [0, 2], {};
        TCvar2: R2, Out, [0, 5], {};
end
```

Figure 23: Formal specification for GRC Vehicle.

```
Class ControllerR [@PL, @QL]
  Events: Near?@PL, TurnOn!@QL,
     GoAhead!@PL, TimedOut, TurnOff!@QL
  States: *idle, activate, busy, monitor, deactivate
  Attributes: inQueue: PLQueue
  Traits: Queue[@PL, PLQueue]
  Attribute-Function:
     idle -> {}; activate -> {inQueue};
     busy -> {inQueue}; monitor -> {inQueue};
     deactivate -> {};
  Transition-Specifications:
     R1: <idle,activate>; Near(true);
        true => inQueue' = append(pid,empty);
     R2: <activate,busy>; TurnOn(true); true => true;
     R3: <activate,activate>; Near(NOT in(pid,inQueue));
        true => inQueue' = append(pid,inQueue);
     R4: <busy,monitor>; GoAhead(pid = head(inQueue));
        true => inQueue' = tail(inQueue);
     R5: <busy,busy>; Near(NOT in(pid,inQueue));
        true => inQueue' = append(pid,inQueue);
     R6: <monitor,deactivate>; TimedOut(true); true => true;
     R7: <monitor,monitor>; GoAhead(pid = head(inQueue));
        true => inQueue' = tail(inQueue);
     R8: <monitor,monitor>; Near(NOT in(pid,inQueue));
        true => inQueue' = append(pid,inQueue);
     R9: <deactivate,idle>; TurnOff(true);
        len(inQueue) = 0 => true;
     R10: <deactivate,activate>; TurnOff(true);
        len(inQueue)>0 => true;
  Time-Constraints:
     TCvar1: R1, TurnOn, [0, 2], {};
     TCvar2: R2, GoAhead, [1, 2], {};
     TCvar3: R2, TimedOut, [10, 40], {};
     TCvar4: R6, TurnOff, [0, 1], {activate};
     TCvar5: R6, TurnOff, [0, 1], {idle};
     TCvar6: R10, TurnOn, [20, 30], {};
end
```

Figure 24: Formal specification for GRC ControllerR.

```
Class ControllerML [@T, @M, @S]
   Events: Near?@T, Req!@M, Grant?@M, TurnOff!@S,
      TurnOn!@S, GoAhead!@T, TimedOut, Ret!@M
   States: *idle, activate, request, deactivate, allocate,
      busy, monitor, release
   Attributes: inQueue:TQueue
   Traits: Queue[@T, TQueue]
   Attribute-Function:
      idle -> {}; activate -> {inQueue};
      request -> {inQueue}; deactivate -> {};
      allocate -> {inQueue}; busy -> {inQueue};
      monitor -> {inQueue}; release -> {};
   Transition-Specifications:
      R1: <idle,activate>; Near(true);
          true => inQueue' = append(pid,empty);
      R2: <activate,request>; Req(true); true => true;
      R3: <activate,activate>; Near(NOT in(pid,inQueue));
          true => inQueue' = append(pid,inQueue);
      R4: <request,request>; Near(NOT in(pid,inQueue));
          true => inQueue' = append(pid,inQueue);
      R5: <request,allocate>; Grant(true); true => true;
      R6: <deactivate,release>; TurnOff(true); true => true;
      R7: <allocate,allocate>; Near(NOT in(pid,inQueue));
          true => inQueue' = append(pid,inQueue);
      R8: <allocate,busy>; TurnOn(true); true => true;
      R9: <busy,monitor>; GoAhead(pid = head(inQueue));
          true => inQueue' = tail(inQueue);
      R10: <busy,busy>; Near(NOT in(pid,inQueue));
          true => inQueue' = append(pid,inQueue);
      R11: <monitor,monitor>; Near(NOT in(pid,inQueue));
          true => inQueue' = append(pid,inQueue);
      R12: <monitor,monitor>; GoAhead(pid = head(inQueue));
          true => inQueue' = tail(inQueue);
      R13: <monitor,deactivate>; TimedOut(true); true => true;
      R14: <release,idle>; TurnOff(true);
          len(inQueue) = 0 => true;
      R15: <release,activate>; TurnOff(true);
          len(inQueue) > 0 => true;
   Time-Constraints:
      TCvar1: R1, Req, [0, 2], {};
      TCvar2: R5, TurnOn, [0, 2], {};
      TCvar3: R5, TimedOut, [10, 40], {};
      TCvar4: R8, GoAhead, [1, 2], {};
      TCvar5: R13, TurnOff, [0, 1], {};
      TCvar6: R6, Ret, [6, 8], {activate};
      TCvar7: R6, Ret, [6, 8], {idle};
      TCvar8: R15, Req, [20, 30], {};
end
```

Figure 25: Formal specification for GRC ControllerML.

```
Class Arbiter [@N]
  Events: Req?@N, Ret?@N, Grant!@N, ToWait
  States: *idle, oneBusy, allocate, twoBusy, busyWait
  Attributes: rQueue:NQueue; hold1:@N; hold2:@N; validity: PToN;
  Traits: Queue[@N,NQueue]; PortIDToNat[PToN];
  Attribute-Function: idle -> {hold1, hold2};
    oneBusy -> {rQueue, hold1, hold2};
    allocate -> {rQueue, hold1, hold2};
    twoBusy -> {rQueue, hold1, hold2};
    busyWait -> {rQueue};
  Transition-Specifications:
    R1: <idle,allocate>; Req(true); true => rQueue' = append(pid,empty);
    R2: <oneBusy,allocate>; Req(true); true => rQueue' = append(pid,rQueue);
    R3: <oneBusy,idle>; Ret(pid = hold1 OR pid = hold2);
        true => hold1' = NULLPORT AND hold2' = NULLPORT;
    R4: <allocate,oneBusy>; Grant(pid = head(rQueue));
        hold1 = NULLPORT AND hold2 = NULLPORT
        AND len(rQueue) = 1 => rQueue' = tail(rQueue) AND hold1' = pid;
    R5: <allocate,twoBusy>; Grant(pid = head(rQueue));
        (hold1 = NULLPORT AND hold2 <> NULLPORT
        AND isValid(hold2,head(rQueue))) OR
        (hold1 <> NULLPORT AND hold2 = NULLPORT
        AND isValid(hold1,head(rQueue))) =>
        (hold1' = pid AND hold2' = hold2) OR (hold2' = pid AND hold1' = hold1);
    R6: <allocate,allocate>; Grant(pid = head(rQueue));
        (hold1 = NULLPORT AND hold2 = NULLPORT)
        AND len(rQueue) > 1 => rQueue' = tail(rQueue) AND hold1' = pid;
    R7: <allocate,busyWait>; ToWait(true);
        (hold1 = NULLPORT AND hold2 <> NULLPORT
        AND NOT isValid(hold2,head(rQueue))) OR
        (hold1 <> NULLPORT AND hold2 = NULLPORT
        AND NOT isValid(hold1,head(rQueue))) => true;
    R8: <twoBusy,oneBusy>;
        Ret(pid = hold1 OR pid = hold2); len(rQueue) = 0 =>
        (hold1' = NULLPORT AND hold2' = hold2) OR
        (hold2' = NULLPORT AND hold1' = hold1);
    R9: <twoBusy,allocate>;
        Ret(pid = hold1 OR pid = hold2); len(rQueue) > 0 =>
        (hold1' = NULLPORT AND hold2' = hold2) OR
        (hold2' = NULLPORT AND hold1' = hold1);
    R10: <twoBusy,twoBusy>; Req(NOT in(pid,rQueue));
        true => rQueue' = append(pid,rQueue);
    R11: <busyWait,allocate>; Ret(pid = hold1 OR pid = hold2);
        true => hold1' = NULLPORT AND hold2' = NULLPORT;
    R12: <busyWait,busyWait>; Req(NOT in(pid,rQueue));
        true => rQueue' = append(pid,rQueue);
  Time-Constraints:
end
```

Figure 26: Formal specification for GRC Arbiter.

```
Subsystem NorthBoundR
    Include:
    Instantiate:
        V1::Vehicle[@P:1];
        V2::Vehicle[@P:1];
        V3::Vehicle[@P:1];
        CR1::ControllerR[@PL:3, @QL:1];
        LR1::Light[@C:1];
    Configure:
        LR1.@C1:@C <-> CR1.@QL1:@QL;
        CR1.@PL1:@PL <-> V1.@P1:@P;
        CR1.@PL2:@PL <-> V2.@P2:@P;
        CR1.@PL3:@PL <-> V3.@P3:@P;
end
```

Figure 27: Formal Specification for Subsystem NorthBoundR.

```
Subsystem NorthBoundL
    Include:
    Instantiate:
        V4::Vehicle[@P:1];
        V5::Vehicle[@P:1];
        V6::Vehicle[@P:1];
        CL1::ControllerML[@T:3, @M:1, @S:1];
        LL1::Light[@C:1];
    Configure:
        LL1.@C1:@C <-> CL1.@S1:@S;
        CL1.@T1:@T <-> V4.@P4:@P;
        CL1.@T2:@T <-> V5.@P5:@P;
        CL1.@T3:@T <-> V6.@P6:@P;
end
```

Figure 28: Formal Specification for Subsystem NorthBoundL.

```
Subsystem RoadTraffic
    Include:
        NorthBoundR, NorthBoundL, NorthBoundM,
        SouthBoundR, SouthBoundL, SouthBoundM,
        EastBoundR, EastBoundL, EastBoundM,
        WestBoundR, WestBoundL, WestBoundM;
    Instantiate:
        A::Arbiter[@N:8];
    Configure:
        CL1.@M1:@M <-> A.@N1:@N;
        CL2.@M2:@M <-> A.@N2:@N;
        CL3.@M3:@M <-> A.@N3:@N;
        CL4.@M4:@M <-> A.@N4:@N;
        CM1.@M5:@M <-> A.@N5:@N;
        CM2.@M6:@M <-> A.@N6:@N;
        CM3.@M7:@M <-> A.@N7:@N;
        CM4.@M8:@M <-> A.@N8:@N;
end
```

Figure 29: Formal Specification for Subsystem RoadTraffic.

Chapter 5

Formal Semantics of RTUML

In defining formal semantics for RTUML modeling technique, we provide both denotational and operational semantics in the specification language of PVS. The denotational semantics provide a meaning for each valid syntactic construct in the language. Defining denotational semantics can only be achieved formally only with a complete formalization of the notation. In the case of RTUML, this involves formalizing the metamodel, including its abstract syntax and its well-formedness rules. To provide a meaning to syntactic constructs, we need a domain of well-defined concepts that exist in the universe of discourse. We provide this domain in terms of types and concepts related to the abstract reactive object model. Finally, we give the meaning of each construct by defining a mapping from the syntactic constructs to concepts in the semantic domain. We provide operational semantics by formalizing an axiomatization of the abstract reactive object model.

5.1 Introduction

This chapter is organized as follows. Section 5.2 outlines the steps involved in defining formal denotational semantics for a specification language, and summarizes our approach in defining semantics for RTUML. Section 5.3 includes definitions for the semantic domain, namely, the time domain, the domain of types, and the event and state domains. Section 5.4 includes similar definitions for the generic reactive classes, the extended statechart, and the GRC type. Section 5.5 defines the concepts of a configuration, a scenario, and a reactive system model. Section 5.6 describes the semantic mapping provided from RTUML model elements to semantic domain concepts. Section 5.7 presents an operational semantics in OCL for the abstract reactive model, and indicates how it is embedded in the proposed semantic domain.

5.2 Denotational Semantics

In this section, we outline the steps in defining semantics for RTUML. We adopt the procedure described by Rumpe [Rum98], and describe

- the subset of the notation being considered.
- the extensions introduced to this subset of the notation,
- the application domain and its characteristics, and
- the relationship between the constructs of the notation and the concepts from the application domain.

Methodology for Defining Formal Semantics

A syntax defines a language $\mathcal{L}$: while in textual notations the syntax involves a linear sequence of characters, in graphic notations the syntax involves a set of diagrams. The steps in defining the syntax of a textual notation are:

- define the alphabet as a set of characters.
- define the lexical syntax by grouping the characters into words using regular expressions.
- define the abstract syntax by grouping the words into sentences using a context free grammar, and
- constrain the sentences using context conditions.

In a graphic notation, lines and characters correspond to the alphabet, boxes and arcs correspond to the lexical syntax, and different shapes of boxes and arcs correspond to the abstract syntax. The textual attributes of the boxes and arcs can be defined using a constraint language. A notation $\mathcal{N}_{\mathcal{L}}$ is needed for defining the language $\mathcal{L}$. This definition hints to an algorithm for parsing the language $\mathcal{L}$.

In UML, the metamodel replaces the context-free syntax; UML class diagrams describe the UML abstract syntax. This recursive definition is not appropriate for defining semantics. OCL (Object Constraint Language) expressions and natural language statements on the metamodel define the context conditions for UML. Context conditions specify well-formedness rules that constrain the syntax. Abstract syntax trees and metamodels both serve to define the abstract syntax of a language. However, abstract syntax trees are hierarchical, whereas metamodels are relation-based and do not have a canonical point from where to initiate a semantic definition.

Defining semantics involves defining a semantic domain $\mathcal{S}$ of known concepts, and a mapping $\mathcal{M}$ from the syntax to the semantic domain. The semantic domain indicates the concepts of the application domain including the model of a system. A notation $\mathcal{N}_\mathcal{S}$ is needed for defining the semantic domain $\mathcal{S}$. The semantic mapping $\mathcal{M}$ relates syntactic constructs with concepts of the semantic domain.

$$\mathcal{M} : \mathcal{L} \rightarrow \mathcal{S}.$$

A notation $\mathcal{N}_\mathcal{M}$ is needed to describe the mapping $\mathcal{M}$. The notation $\mathcal{N}_\mathcal{M}$ must include the notations $\mathcal{N}_\mathcal{L}$ and $\mathcal{N}_\mathcal{S}$ used for describing the syntactic and semantic domains.

$$\mathcal{N}_\mathcal{L}, \mathcal{N}_\mathcal{S} \subseteq \mathcal{N}_\mathcal{M}.$$

The benefits of a formal definition of the semantic mapping include that the mapping supports reasoning. If a formal notation is used as the semantic domain $\mathcal{S}$ and the mapping $\mathcal{M}$ is defined in an algorithmic style, then documents in the informal language $\mathcal{L}$ can be translated into documents in the formal notation $\mathcal{S}$ for rigorous analysis. For instance, a proof of consistency may be provided to ensure that documents in the language $\mathcal{L}$ are consistent models. In case a proof of consistency cannot be provided for the language $\mathcal{L}$, then context conditions must be introduced on the syntax of $\mathcal{L}$ so that it is constrained in such a way that the proof of consistency holds for every document in the new version of $\mathcal{L}$.

Defining RTUML Formal Semantics

We provide formal denotational semantics for the RTUML modeling technique in the specification language of PVS. Our approach to defining RTUML formal semantics involves the following steps.

1. Formalize the language $\mathcal{L}$.
   - Formalize UML metamodel, with restriction to the packages *Foundation* and *Behavioral Elements*. This involves introducing a type definition for each model element, and predicates and lemmas for the well-formedness rules.
   - Introduce the stereotype extensions in the formal definition of the metamodel, by augmenting the respective set of stereotypes for the model elements *Class* and *Association*.
   - Introduce the RTUML well-formedness rules by providing predicates and lemmas defining restrictions on the UML model elements. This involves translating the OCL specification of the rules given in Section 4.3.
2. Formalize the semantic domain $\mathcal{S}$.
   - Define the concepts in the abstract reactive system model, and identify the constraints on these concepts.
   - Introduce a type definition for each concept, and axioms characterizing the concepts.

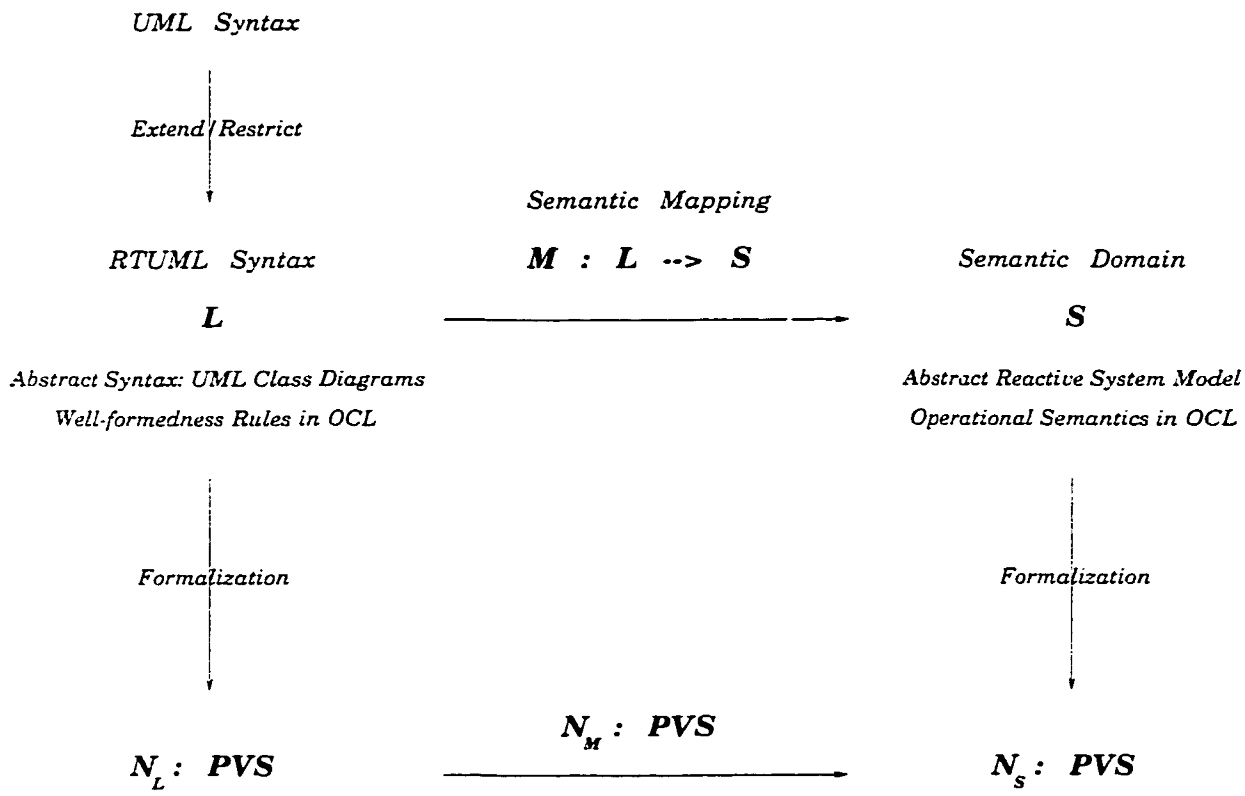


Figure 30: Formal Semantics for RTUML.

3. Formalize the semantic mapping $\mathcal{M}$.

    - Define the relations between the RTUML syntactic constructs (in terms of the model elements) and the concepts in the semantic domain. This involves matching every RTUML well-formed construct with a concept in the semantic domain.
    - For each RTUML well-formed construct, define a function that maps the construct to the semantic domain concept.
    - Introduce axioms to characterize the functions. The semantic mapping corresponds to the union of the set of functions.

Figure 30 illustrates the process of defining formal semantics for RTUML. *Semantic completeness* for RTUML is ensured by providing a function for each well-formed construct in the language. Consequently, every well-formed RTUML construct has a *meaning*. For a given well-formed RTUML construct and the corresponding semantic domain concept, applying the respective function to a PVS specification of the construct yields the PVS specification of the concept. Appendix A includes a list of tables describing the formalization of UML metamodel, and the RTUML well-formedness rules. The tables establish the correspondence between the UML packages and the PVS theories. Appendix B contains the specifications relating to the formalization of UML abstract syntax and well-formedness rules; it includes the RTUML stereotype extensions. Appendix C contains the specifications for the RTUML well-formedness rules. Appendix D contains the specifications for the definitions of the concepts in the semantic domain. Appendix E contains the specifications for the functions comprising the semantic mapping, and the axioms characterizing the functions.

Consistency Checking in Design Specifications

Corresponding to each design specification in UML, we can use the formal semantics to formulate a corresponding PVS specification. A relationship R between two UML design components is stated in the form of a set of theorems in a parameterized theory T_R. The theorems in theory T_R instantiated with two actual design specifications must be proved in order to establish the consistency between two designs. The following is a more formal definition of consistency for design specifications.

We define *consistency* between design specifications as follows: let d_1 and d_2 be two design specifications in UML; let p_1 and p_2 be their corresponding PVS specifications. Corresponding to the relationship R between the designs d_1 and d_2, there exists a parameterized theory T_R. If a proof can be constructed for every theorem in the instance $T_R(p_1, p_2)$ of the theory, then design specifications d_1 and d_2 are consistent.

Checking for consistency of design specifications may not be possible without sufficient axioms capturing properties of data types used in the specifications. Consequently, consistency cannot be assured without completeness of abstract data types.

5.2.1 Formalizing UML Metamodel

A suitable approach to formalizing UML metamodel is to identify attributes and properties of model elements relevant to the application domain, and describe their meaning in a mathematical notation. By so doing, we must ensure the *completeness* of the semantics; that is, we must describe the meaning of sufficient attributes and properties of model elements to allow a precise understanding of the structure and behavior of a UML-defined object. This may involve the construction of a formal mathematical object constraint language. Appendix A provides a series of tables describing how each package describing the UML metamodel is translated into a corresponding PVS theory. Appendix B gives the PVS specification of the UML abstract syntax and well-formedness rules. Appendix C gives the PVS specification of the RTUML well-formedness rules.

Mapping UML Notation onto PVS Constructs

We adopt the following steps in formalizing UML abstract syntax.

1. Identify the model elements in each of the selected components of UML notation. UML model elements are described using UML class diagrams with only class name and attributes. However, a metaclass can inherit other metaclasses, and can also have a composition aggregation relationship with other classes.

2. Specify PVS type definitions for each UML model element, using tuple, record, and uninterpreted type definitions. We flatten the hierarchical structure of the abstract syntax to obtain all the attributes of a model element.

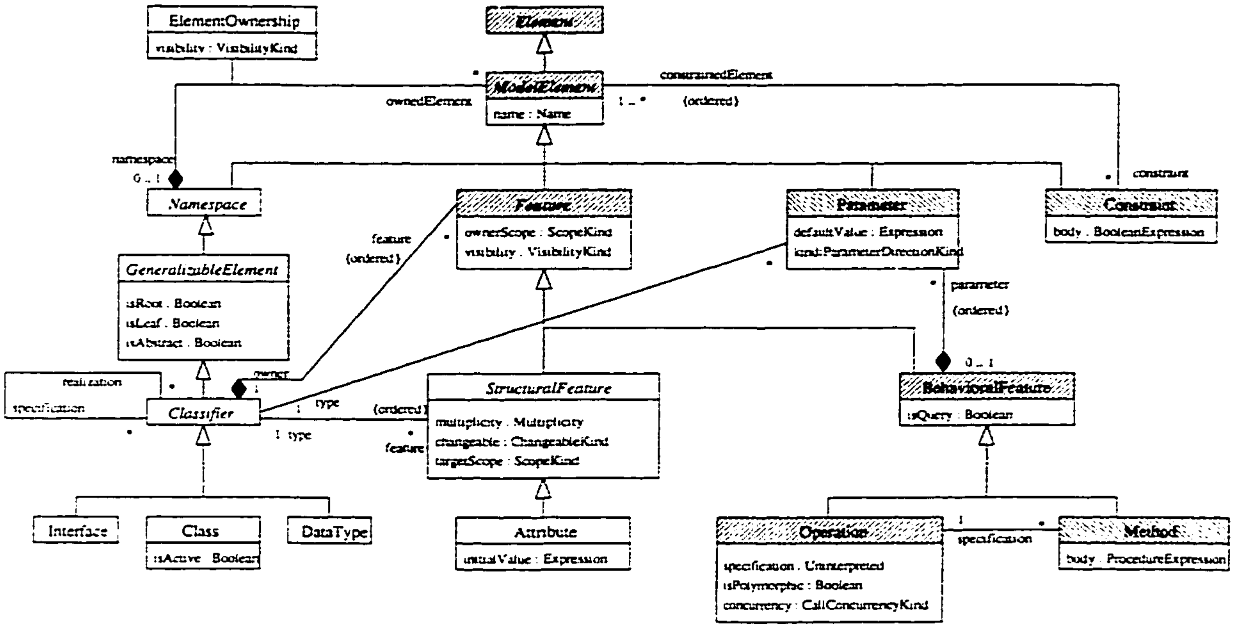


Figure 31: UML Core Package – *Backbone* [OMG99].

Attribute	Type	Inherited from class
specification	Uninterpreted	Operation
isPolymorphic	Boolean	Operation
concurrency	CallConcurrencyKind	Operation
isQuery	Boolean	BehavioralFeature
ownerScope	ScopeKind	Feature
visibility	VisibilityKind	Feature
name	Name	ModelElement

Table 3: Attributes of the Metaclass *Operation*.

3. Give a PVS specification for each well-formedness rule for the metaclass describing a model element. We use predicates and lemmas to formulate invariants and constraints on the model element. These will need to be proved to check the well-formedness of any diagram given in terms of instances of the model element. We use the flattened hierarchical structure of the abstract syntax to obtain all the well-formedness rules corresponding to a model element.

We illustrate this exercise of mapping UML notation onto PVS constructs with an example. The abstract syntax for UML logical package *core* is given in five class diagrams, one for each subpackage. Figure 31 [OMG99] shows the class diagram for the subpackage *Backbone*. The names of abstract metaclasses are shown in italics in the metamodel; the names of metaclasses describing model elements are shown in normal font. The model elements in *Backbone* are *Interface*, *Class*, *DataType*, *Attribute*, *Operation*, *Method*, *Parameter*, *Constraint*, and *ElementOwnership*. The model elements in *Relationships* are *Dependency*, *Generalization*, *AssociationEnd*, *Association*, and *AssociationClass*.

```
operation: TYPE = [# specification: Uninterpreted,
                     isPolymorphic: Boolean,
                     concurrency: CallConcurrencyKind,
                     isQuery: Boolean,
                     ownerScope: ScopeKind,
                     visibility: VisibilityKind,
                     name: Name,
                     parameters: ParameterList,
                     specificationOf: MethodList,
                     constraints: ConstraintList #]
```

Figure 32: PVS Record Type Definition for the Metaclass *Operation*.

We choose the metaclass *Operation* to show how to obtain the attributes of a model element. The class *Operation* inherits the classes *BehavioralFeature*, *Feature*, *ModelElement*, and *Element*. The attributes of the class *Operation* are given in Table 3. In addition, the class *BehavioralFeature* is an aggregation of zero or more *Parameter* objects. There is a *one-to-many* association between the class *Operation* and the class *Method*, and a *many-to-many* association between the class *ModelElement*, inherited by the class *Operation*, and the class *Constraint*. These features can be captured by the following PVS record type definition.

We define static semantics for the model element *Operation* by translating the well-formedness rules for the class *Operation*, and those for the classes inherited by the class *Operation*, into PVS axioms, functions, lemmas and theorems. For instance, one of the rules for the class *Operation* is:

All Parameters should have a unique name.

In OCL, this constraint is specified as follows.

```
self.parameter-> forAll(p1.p2 | p1.name = p2.name implies p1 = p2)
```

We translate this rule into the following PVS predicate.

```
operationPredicate((op: Operation)): bool =
  FORALL (p1,p2: Parameter):
    member(p1,parameters(op)) AND member(p2,parameters(op)) AND
      name(p1) = name(p2) IMPLIES p1 = p2
```

We then include the following lemma to capture this rule: the lemma needs to be proved to ascertain that an operation is consistently defined in a UML diagram.

```
operationRule: LEMMA
  FORALL (op: Operation): operationPredicate(op)
```

5.3 Semantic Domain

We define domain in terms of concepts from the abstract reactive object model described in Chapter 2. The main components include generic reactive classes, configurations, and scenarios. We use a domain of types to capture generic reactive classes, port types and data types. Appendix D provides the PVS specification of the semantic domain.

5.3.1 Time Domain

- *Time* is defined as the set of natural numbers.

$$Time = \{t \mid t \in Nat\} \cup \{\infty\}.$$

5.3.2 Domain of Types

- GRCTypes
  - $\mathcal{GRC}$ is the universal set of GRCTypes.
- DataTypes
  - $\mathcal{D}$ is the universal set of DataTypes.
- PortTypes
  - $\mathcal{P}$ is the universal set of PortTypes.
  - P_0 is the *null PortType*.
  * $\mathcal{P}$ includes the *null PortType* P_0.

$$P_0 \in \mathcal{P}.$$

  * P_0 is a *singleton* set, containing the *null port* p_0.

$$P_0 = \{p_0\}.$$

* Notation

  For every type T, there exists a maximal set S, such that every element of S is of type T. We use T to denote both the type T and the maximal set associated with the type T. Consequently, $a : T$ denoting a is an *instance* of T, holds iff a is a member of the maximal set associated with the type T.

$$a : T \leftrightarrow a \in T.$$

5.3.3 Event Domain

- $\mathcal{E}$ is the universal set of events.
- $\mathcal{E}_{internal}$ is the universal set of *internal* events.
- $\mathcal{E}_{external}$ is the universal set of *external* events.

- $\mathcal{E}_{input}$ is the universal set of *input* events. An *input* event corresponds to an *external* event suffixed with the symbol "?". We use "e?" to denote the *input* event obtained by suffixing the *external* event "e".

$$\mathcal{E}_{input} = \{e? \mid e \in \mathcal{E}_{external}\}.$$

- $\mathcal{E}_{output}$ is the universal set of *output* events. An *output* event corresponds to an *external* event suffixed with the symbol "!". We use "e!" to denote the *output* event obtained by suffixing the *external* event "e".

$$\mathcal{E}_{output} = \{e! \mid e \in \mathcal{E}_{external}\}.$$

The following properties apply to the universal sets of *events*.

1. The universal sets of *internal* and *external* events are disjoint.

$$\mathcal{E}_{internal} \cap \mathcal{E}_{external} = \emptyset.$$

2. The universal sets of *internal* and *input* events are disjoint.

$$\mathcal{E}_{internal} \cap \mathcal{E}_{input} = \emptyset.$$

3. The universal sets of *internal* and *output* events are disjoint.

$$\mathcal{E}_{internal} \cap \mathcal{E}_{output} = \emptyset.$$

4. The universal sets of *external* and *input* events are disjoint.

$$\mathcal{E}_{external} \cap \mathcal{E}_{input} = \emptyset.$$

5. The universal sets of *external* and *output* events are disjoint.

$$\mathcal{E}_{external} \cap \mathcal{E}_{output} = \emptyset.$$

6. The universal sets of *input* and *output* events are disjoint.

$$\mathcal{E}_{input} \cap \mathcal{E}_{output} = \emptyset.$$

7. The universal set of events $\mathcal{E}$ corresponds to the union of the universal sets of *internal*, *input* and *output* events.

$$\mathcal{E} = \mathcal{E}_{internal} \cup \mathcal{E}_{input} \cup \mathcal{E}_{output}.$$

The following functions are defined on the *Event* domain.

* The bijective function σ_{in} converts a set of *input* events into a set of *external* events, by removing the suffix "?" from each *input* event.

$$\sigma_{in} : \wp(\mathcal{E}_{input}) \rightarrow \wp(\mathcal{E}_{external}).$$

  * The function σ_{in} is defined as follows.

$$\forall E_{in} \in \wp(\mathcal{E}_{input}) \bullet \sigma_{in}(E_{in}) = \{e \mid e? \in E_{in}\}.$$

* The bijective function σ_{out} converts a set of *output* events into a set of *external* events, by removing the suffix "!" from each *output* event.

$$\sigma_{out} : \wp(\mathcal{E}_{output}) \rightarrow \wp(\mathcal{E}_{external}).$$

  * The function σ_{out} is defined as follows.

$$\forall E_{out} \in \wp(\mathcal{E}_{output}) \bullet \sigma_{out}(E_{out}) = \{e \mid e! \in E_{out}\}.$$

5.3.4 State Domain

- $\mathcal{S}$ is the universal set of states.
- $\mathcal{S}_{simple}$ is the universal set of *simple* states.
- $\mathcal{S}_{complex}$ is the universal set of *complex* states.

The following properties apply to the universal sets of *states*.

1. The universal sets of *simple* and *complex* states are disjoint.

$$\mathcal{S}_{simple} \cap \mathcal{S}_{complex} = \emptyset.$$

2. The universal set of states $\mathcal{S}$ corresponds to the union of the universal sets of *simple* and *complex* states.

$$\mathcal{S} = \mathcal{S}_{simple} \cup \mathcal{S}_{complex}.$$

The following functions are defined on the *State* domain.

* A *complex* state contains a set of *substates* that are either simple or complex. The function *substates* returns the set of substates of a state.

$$substates : \mathcal{S} \rightarrow \wp(\mathcal{S}).$$

  1. The function *substates* returns an empty set when applied to a *simple* state.

$$\forall s \in \mathcal{S}_{simple} \bullet substates(s) = \emptyset.$$

2. The set of *substates* of a *complex* state is nonempty.

$$\forall\, s \in S_{complex} \bullet substates(s) \neq \emptyset.$$

* A complex state has a unique *entry* state, which is a simple state. The total function *entry* identifies the entry state of a complex state.

$$entry: S_{complex} \rightarrow S_{simple}.$$

1. The *entry* state of a complex state is a member of the *substates* of the complex state.

$$\forall\, s \in S_{complex} \bullet entry(s) \in substates(s).$$

2. The set of *substates* of a complex state contains at least 2 elements.

$$\forall\, s \in S_{complex} \bullet substates(s) \setminus \{\, entry(s)\, \} \neq \emptyset.$$

* The hierarchy function $\mathcal{H}$ returns the set of all states in the hierarchy of a state.

$$\mathcal{H}: S \rightarrow \wp(S).$$

1. The hierarchy function $\mathcal{H}$ is recursively defined as follows.

$$\mathcal{H}(s) = \{\, s\,\} \cup \bigcup_{v \in substates(s)} \mathcal{H}(v).$$

5.4 Reactive Object Model – *GRCType*

The reactive object model consists of a generic reactive class with an associated statechart extended to capture timing constraints on transitions. This definition captures our notion of what a reactive object is.

5.4.1 Definition for *GRCClass*

A *GRCClass* is a 3-tuple $< P, A, \Omega >$, with the following definition.

- P is a set of *PortTypes*.

$$P: \wp(\mathcal{P}).$$

- A is a set of *attributes*, where each attribute is either an instance of a PortType from the set P, or an instance of a DataType from the universal set $\mathcal{D}$.

$$A = \{a \mid a: P\} \cup \{a \mid a: \mathcal{D}\}.$$

- Ω is a mapping from the set of PortTypes P to the power set of the universal set of events $\mathcal{E}$, defining the set of *allowed* input and output events at instances of the PortType.

$$\Omega: P \rightarrow \wp(\mathcal{E}).$$

The following properties apply to a *GRC* class $< P, A, \Omega >$.

1. The sets of *input* and *output* events associated with a PortType, other than the *null PortType* P_0, correspond to disjoint sets of *external* events.

$$\forall P_i \in P \bullet P_i \neq P_0 \rightarrow$$

$$\Omega(P_i) = E_{in_i} \cup E_{out_i} \wedge E_{in_i} \subseteq \mathcal{E}_{input} \wedge E_{out_i} \subseteq \mathcal{E}_{output} \wedge$$

$$\sigma_{in}(E_{in_i}) \cap \sigma_{out}(E_{out_i}) = \emptyset.$$

2. The sets of events associated with two PortTypes of a GRC class are disjoint.

$$\forall P_i, P_j \in P \bullet P_i \neq P_j \rightarrow \Omega(P_i) \cap \Omega(P_j) = \emptyset.$$

3. The events associated with the *null PortType* P_0 are *internal* events.

$$\forall P_i \in P \bullet P_i = P_0 \rightarrow \Omega(P_i) \subseteq \mathcal{E}_{internal}.$$

4. The set of *internal* events $E_{internal}$ of a GRC class corresponds to the set of events associated with the *null PortType* P_0.

$$P_0 \in P \rightarrow E_{internal} = \Omega(P_0) \wedge P_0 \notin P \rightarrow E_{internal} = \emptyset.$$

5. The events associated with a PortType P_i of a GRC class, excluding the *null PortType* P_0, are *input* and *output* events.

$$\forall P_i \in P \bullet P_i \neq P_0 \rightarrow \Omega(P_i) \subseteq (\mathcal{E}_{input} \cup \mathcal{E}_{output}).$$

6. The set of *input* and *output* events E_{io} of a GRCClass corresponds to the distributed union of the sets of events associated with the PortTypes of the GRCClass, excluding the *null PortType* P_0.

$$E_{io} = \bigcup_{P_i \in P \wedge P_i \neq P_0} \Omega(P_i).$$

Compatibility of PortTypes

Given two *GRCClasses* G_a and G_b, such that

$$G_a = < P_a, A_a, \Omega_a >, \text{ and}$$

$$G_b = < P_b, A_b, \Omega_b >,$$

PortType P_{a_i} from the set P_a and PortType P_{b_j} from the set P_b are *compatible* iff the set of *input* events associated with PortType P_{a_i} is equal to the set of *output* events associated with PortType P_{b_j}, and the set of *output* events associated with PortType P_{a_i} is equal to the set of *input* events associated with PortType P_{b_j}. The comparison for equality of events is applied to the *external* events corresponding to the respective *input* and *output* events.

- The predicate *compatible* defines the compatibility of two PortTypes.

$$compatible: \mathcal{P} \times \mathcal{P} \rightarrow Bool.$$

1. The predicate *compatible* has the following definition.

$$\forall P_{a_i} \in P_a; P_{b_j} \in P_b \bullet compatible(P_{a_i}, P_{b_j}) \leftrightarrow$$

$$\Omega_a(P_{a_i}) = E_{in_a} \cup E_{out_a} \wedge E_{in_a} \subseteq \mathcal{E}_{input} \wedge E_{out_a} \subseteq \mathcal{E}_{output} \wedge$$

$$\Omega_b(P_{b_j}) = E_{in_b} \cup E_{out_b} \wedge E_{in_b} \subseteq \mathcal{E}_{input} \wedge E_{out_b} \subseteq \mathcal{E}_{output} \wedge$$

$$\sigma_{in}(E_{in_a}) = \sigma_{out}(E_{out_b}) \wedge \sigma_{in}(E_{in_b}) = \sigma_{out}(E_{out_a}).$$

2. Two PortTypes of the same GRCClass are not *compatible*.

$$\forall P_{a_i}, P_{a_j} \in P_a \bullet \neg\, compatible(P_{a_i}, P_{a_j}).$$

5.4.2 Definition for *Extended Statechart*

An *ExtendedStatechart* is a 8-tuple $< S, C, R, \Upsilon, \Phi, \Psi, \Gamma, \Xi >$, with the following definition.

- S is a set of *states*.

$$S: \wp(S).$$

1. The set S consists of *simple* and *complex* states. The set S_s denotes the subset of all simple states from the set S, and the set S_c denotes the subset of all complex states from the set S.

$$S = S_s \cup S_c \wedge S_s \subseteq S_{simple} \wedge S_c \subseteq S_{complex}.$$

2. The distinguished simple state s_0 from the set S is the *initial* state of the statechart.

$$s_0 \in S_s.$$

3. The *entry* state of a *complex* state from the set S_c is a *simple* state from the set S_s.

$$\forall s_c \in S_c \bullet entry(s_c) \in S_s.$$

4. The set of *substates* of a complex state from the set S_c is a subset of the set S.

$$\forall s_c \in S_c \bullet substates(s_c) \subset S.$$

5. The sets of *substates* of two complex states from the set S_c are disjoint.

$$\forall s_i, s_j \in S_c \bullet s_i \neq s_j \rightarrow substates(s_i) \cap substates(s_j) = \emptyset.$$

- C is a set of *logical clocks* for defining *time constraints* on reactions to transitions.
- R is a set of *transitions*, where each transition is a 5-tuple $< s, d, e, g, a >$, such that

- s is the *source* state; s is a member of the set S.

$$s \in S.$$

- d is the *destination* state; d is a member of the set S.

$$d \in S.$$

- e is the *trigger* event; e is a member of the universal set of events $\mathcal{E}$.

$$e \in \mathcal{E}.$$

- g is a predicate representing the *guard* condition for the transition.
- a is a predicate representing the *action* resulting from the transition.

1. A valid transition satisfies one of the following four conditions.

   ⋆ The destination state d is a state which is not a substate of another state.

   $$d \in S \setminus \bigcup_{s_c \in S_c} substates(s_c).$$

   ⋆ The destination state d is the *entry* state of a complex state which is not a substate of another state.

   $$\exists\, d_c \in S_c \setminus \bigcup_{s_c \in S_c} substates(s_c) \bullet d = entry(d_c).$$

   ⋆ There exists a complex state s_c such that the source state s is in the hierarchy of the state s_c and the destination state d is a substate of the state s_c.

   $$\exists\, s_c \in S_c \bullet s \in \mathcal{H}(s_c) \wedge d \in substates(s_c).$$

   ⋆ There exists a complex state s_c such that the source state s is in the hierarchy of the state s_c and the destination state d is the *entry* state of a complex substate of the state s_c.

   $$\exists\, s_c, d_c \in S_c \bullet s \in \mathcal{H}(s_c) \wedge d_c \in substates(s_c) \wedge d = entry(d_c).$$

2. If the source state s is a complex state, then the transition corresponds to a set of transitions R_s such that the source state s_h, of each transition in the set R_s, is in the hierarchy of the state s.

   $$s \in S_c \rightarrow \{< s_h, d, e, g, a > \mid s_h \in \mathcal{H}(s)\} \subseteq R.$$

3. If the destination state d is a complex state, then the transition corresponds to a transition whose destination state d_e is the entry state of the state d.

   $$d \in S_c \rightarrow \{< s, d_e, e, g, a > \mid d_e = entry(d)\} \subseteq R.$$

- Υ is a total function that maps a clock from the set C to a transition from the set R, such that the clock is initialized to 0 when the transition occurs.

$$\Upsilon : C \rightarrow R.$$

- Φ is a function that maps a transition from the set R to a set of clocks from the set C, such that the timing constraint on the transition is specified in terms of the values of these clocks.

$$\Phi : R \rightarrow \wp(C).$$

1. If the trigger event e of the transition $r = \,<s,d,e,g,a>$ is an *input* event, then the transition r is not time-constrained.

$$\forall\, r \in R \bullet r = \,<s,d,e,g,a> \;\wedge\; e \in \mathcal{E}_{input} \rightarrow \Phi(r) = \emptyset.$$

2. If the transition $r = \,<s,d,e,g,a>$ is time-constrained, then the trigger event e is either an *internal* or an *output* event.

$$\forall\, r \in R \bullet r = \,<s,d,e,g,a> \;\wedge\; \Phi(r) \neq \emptyset \rightarrow e \in (\mathcal{E}_{internal} \cup \mathcal{E}_{output}).$$

3. If the transition $r = \,<s,d,e,g,a>$ is time-constrained, then the trigger event e corresponds to a reaction to transitions other than r from the set R, such that the corresponding clocks are initialized to 0 when those transitions occur.

$$\forall\, r \in R \bullet \Phi(r) \neq \emptyset \rightarrow \forall\, c \in \Phi(r) \bullet \exists\, r_2 \in R \bullet \Upsilon(c) = r_2 \wedge r_2 \neq r.$$

- Ψ is a partial function that gives the value of a clock from the set C when a transition from the set R occurs.

$$\Psi : C \times R \rightarrow Time.$$

  * The function Ψ is *defined* on the clock c and the transition r iff
    * the clock c is initialized to 0 when the transition r occurs, or
    * the transition r corresponds to a time-constrained reaction, such that the timing constraint is defined in terms of the clock c, or
    * the destination state of the transition r is a *disabling state* for a time-constrained reaction, such that the timing constraint is defined in terms of the clock c.

$$\Psi(c,\, r) = \begin{cases} 0, & \text{if } \Upsilon(c) = r, \\ x, \text{ where } 0 < x < \infty, & \text{if } c \in \Phi(r), \\ \infty, & \text{if } r = \,<s,d,e,g,a> \;\wedge\; d \in \Xi(c). \end{cases}$$

1. If transition r corresponds to a time-constrained reaction to some other transition r_2 from the set R, and the timing constraint on the reaction is defined in terms of a clock from the set C, then the value of the clock is initialized to 0 when transition r_2 occurs.

$$\forall\, c \in \Phi(r) \bullet \Upsilon\,(c) = r_2 \rightarrow \Psi(c,\, r_2) = 0.$$

- Γ is a partial function that gives the *lower* and *upper bounds* for the value of a clock from the set C, relative to the activation time of the time-constrained reaction, within which the corresponding transition from the set R can occur.

$$\Gamma : C \times R \rightarrow Time \times Time.$$

  * The function Ψ is *defined* on the clock c and the transition r iff the transition r corresponds to a time-constrained reaction, such that the timing constraint is defined in terms of the clock c.

$$\Gamma(c,\, r) = \,<l,u> \quad \text{iff } c \in \Phi(r) \wedge l < u.$$

- Ξ is a function that maps a clock c from the set C to the set of *disabling* states for the time-constrained reaction defined in terms of the clock c.

$$\Xi : C \rightarrow \wp(S).$$

5.4.3 Definition for *GRCType*

A *GRCType* is a 2-tuple $< G, B >$, with the following definition.

- G is a *GRCClass*.

$$G = \ < P, A, \Omega > .$$

- B is an *ExtendedStatechart*.

$$B = \ < S, C, R, \Upsilon, \Phi, \Psi, \Gamma, \Xi > .$$

- A transition r from the set R, such that $r = \ < s, d, e, g, a >$, satisfies the following properties.

  1. The *trigger* event e is associated with a PortType from the set of PortTypes P.

  $$\exists \, P_i \in P \bullet e \in \Omega(P_i).$$

  * The *guard* condition g is a *conjunction* of three predicates g_{port_cond}, $g_{enabling\_cond}$ and $g_{time\_constraint}$.

  $$g = g_{port\_cond} \wedge g_{enabling\_cond} \wedge g_{time\_constraint}.$$

  ⋆ The *port condition* g_{port_cond} is a logical assertion on the values of *attributes* from the set A and an instance p_j of a PortType P_i from the set P. The PortType P_i corresponds to the PortType with which event e is associated; the *null PortType* P_0 for internal events. The instance p_j of PortType P_i corresponds to the port through which the message is channeled; the *null port* p_0 for internal events.

  $$g_{port\_cond} : \wp(A) \times P_i \rightarrow Bool.$$

  where

  $$P_i \in P \wedge e \in \Omega(P_i).$$

  2. If the trigger event e is an *internal* event, the *port condition* is *true*.

  $$\text{if } e \in \mathcal{E}_{internal} \text{ then}$$
  $$g_{port\_cond} \triangleq true.$$

  ⋆ The *enabling condition* $g_{enabling\_cond}$ is a logical assertion on the values of *attributes* from the set A.

  $$g_{enabling\_cond} : \wp(A) \rightarrow Bool.$$

  ⋆ The *time constraint condition* $g_{time\_constraint}$ is a logical assertion on the values of *clocks* from the set C, that are used in specifying the timing constraint on transition r.

  $$g_{time\_constraint} : \wp(C) \rightarrow Bool.$$

- ⋆ The *time constraint condition* $g_{time\_constraint}$ is a distributed disjunction specifying the *lower* and *upper* bounds l_c and u_c on the value of each *clock* c that is used in specifying the timing constraint on transition r.

$$g_{time\_constraint} \triangleq \bigvee_{c \in \Phi(r)} \Psi(c, r) > (\Gamma(c, r))'1 \wedge \Psi(c, r) < (\Gamma(c, r))'2.$$

3. If there is no timing constraint on transition r, the *time constraint condition* is *true*.

$$\text{if } \Phi(r) = \emptyset \text{ then}$$
$$g_{time\_constraint} \triangleq true.$$

4. If the trigger event e is an *input* event, the *time constraint condition* is *true*.

$$\text{if } e \in \mathcal{E}_{input} \text{ then}$$
$$g_{time\_constraint} \triangleq true.$$

- ⋆ The *action* a is a *conjunction* of two predicates a_{post_cond} and a_{clock_init}.

$$a = a_{post\_cond} \wedge a_{clock\_init}.$$

- ⋆ The *post condition* a_{post_cond} is a logical assertion on the values of *attributes* from the set A after transition r is taken.

$$a_{post\_cond} : \wp(A) \rightarrow Bool.$$

- ⋆ The *clock initialization expression* a_{clock_init} is a logical assertion on the values of *clocks* from the set C, to initialize the value of a clock to 0 for each *time-constrained* reaction associated with transition r.

$$a_{clock\_init} : \wp(C) \rightarrow Bool.$$

- ⋆ The *clock initialization expression* a_{clock_init} initializes the value of a *clock* to 0 for each *time-constrained* reaction associated with transition r.

$$a_{clock\_init} \triangleq \bigwedge_{c \in C \wedge \Upsilon(c) = r} \Psi(c, r) = 0.$$

- For each state s from the set S, the *disabling state expression* $e_{disabling\_state}$ defines the time-constrained reactions that are disabled when a transition leading to the state s occurs. The *disabling state expression* $e_{disabling\_state}$ is defined as a predicate on clocks from the set C that are used for specifying the timing constraints on the reactions.

$$e_{disabling\_state} : \wp(C) \rightarrow Bool.$$

- ⋆ The *disabling state expression* $e_{disabling\_state}$ sets the values of the clocks defining timing constraints on reactions that are disabled in state s to infinity (∞).

$$e_{disabling\_state} \triangleq \bigwedge_{c \in C \wedge d \in \Xi(c) \wedge r \in R \wedge r = <s,d,e,g,a>} \Psi(c, r) = \infty.$$

5.5 Reactive System Model

The reactive system model consists of generic reactive classes each with an associated extended statechart, configurations of instances of reactive object models and scenarios of interaction among the reactive objects. This definition captures our notion of what a reactive system is.

5.5.1 Definition for *Configuration*

A *configuration* is a 4-tuple $< V, I, W, L >$, with the following definition.

- V is a set of *reactive objects*.

  1. A reactive object from the set V is an instance of a GRCType.

     $\forall\, v \in V \bullet \exists\, G \in \mathcal{GRC} \bullet v : G.$

- I is a set of *port objects*.

  1. A port object from the set I is an instance of a PortType.

     $\forall\, p \in I \bullet \exists\, P_i \in \mathcal{P} \bullet p : P_i.$

- W is a function that defines a set of *port ownership* associations, identifying port objects from the set I that are owned by a reactive object from the set V.

  $W : V \rightarrow \wp(I).$

  1. A port object from the set I is owned by a unique reactive object from the set V. The sets of port objects associated with two reactive objects are disjoint.

     $\forall\, r_i, r_j \in V \bullet r_i \neq r_j \rightarrow W(r_i) \cap W(r_j) = \emptyset.$

- L is a partial injective function that defines a set of *communication channels* between port objects from the set I.

  $L : I \rightarrowtail I.$

  1. A port object from the set I is linked to at most one port object from the set I. The port objects associated with two port objects p_i and p_j are distinct port objects.

     $\forall\, p_i, p_j \in I \bullet p_i \neq p_j \rightarrow L(p_i) \neq L(p_j).$

  2. A *communication channel* between two port objects is bidirectional.

     $\forall\, p_i, p_j \in I \bullet L(p_i) = p_j \leftrightarrow L(p_j) = p_i.$

  3. A *communication channel* associates two port objects only if they are instances of compatible PortTypes.

     $\forall\, p \in I \bullet p \in P_i \wedge L(p) \in P_j \rightarrow compatible(P_i, P_j).$

5.5.2 Definition for *Scenario*

A *scenario* is a 2-tuple $< V, M >$, with the following definition.

- V is a set of reactive objects.

  1. A reactive object from the set V is an instance of a GRCType.

     $\forall v \in V \bullet \exists G \in \mathcal{GRC} \bullet v : G.$

- M is a sequence of messages. A message is a 4-tuple $< v_s, v_r, e, t >$, such that

  1. the *sender* object v_s is a member of the set V.

     $v_s \in V.$

  2. the *receiver* object v_r is a member of the set V.

     $v_r \in V.$

  3. the *event* e is a member of the universal set of events $\mathcal{E}$.

     $e \in \mathcal{E}.$

  * the *time* t corresponds to the time at which the event occurs.

     $t \in Time.$

5.5.3 Definition for *Reactive System Model*

A *ReactiveSystemModel* is a 3-tuple $< Y, F, N >$, with the following definition.

- Y is a set of *GRCTypes*.
- F is a set of *configurations*.
- N is a set of *scenarios*.

The following properties apply to a *ReactiveSystemModel*.

1. In a configuration $f = < V, I, W, L >$ from the set F, a reactive object v from the set V is an instance of a GRCType G from the set Y.

   $\forall < V, I, W, L > \in F \bullet \forall v \in V \bullet \exists G \in Y \bullet v : G.$

2. In a configuration $f = < V, I, W, L >$ from the set F, a port object p from the set I is an instance of a PortType P such that PortType P is owned by a GRCType G from the set Y.

   $\forall < V, I, W, L > \in F \bullet \forall p \in I \bullet \exists < P, A, \Omega > \in Y \bullet p : P.$

3. In a scenario $n = \; < V, M >$ from the set N, a reactive object v from the set V is an instance of a GRCType G from the set Y.

$$\forall < V, M > \; \in N \bullet \forall v \in V \bullet \exists G \in Y \bullet v : G.$$

4. In a scenario $n = \; < V, M >$ from the set N, for every message $m = \; < v_s, v_r, e, t >$ from the set M, there exists a configuration f from the set F, such that port objects p_i and p_j are in the configuration, and there exists GRCTypes G_i and G_j from the set Y, such that PortType P_i is owned by GRCType G_i, PortType P_j is owned by GRCType G_j, port object p_i is an instance of PortType P_i, port object p_j is an instance of PortType P_j, event e is allowed at PortTypes P_i and P_j, and PortTypes P_i and P_j are *compatible*.

$$\forall < V, M > \; \in N \bullet \forall < v_s, v_r, e, t > \; \in M \bullet$$

$$\exists < V, I, W, L > \; \in F, \; p_i, \; p_j \in I \bullet$$

$$\exists < P_a, \; A_a, \; \Omega_a >, \; < P_b, \; A_b, \; \Omega_b > \; \in Y. \; P_i \in P_a, \; P_j \in P_b \bullet$$

$$\exists \, e_{in} \in \Omega_a(P_i), \; e_{out} \in \Omega_b(P_j) \bullet$$

$$p_i : P_i \wedge p_j : P_j \wedge e \in \sigma_{in}(\{e_{in}\}) \wedge e \in \sigma_{out}(\{e_{out}\}) \wedge compatible(P_i, \; P_j).$$

5.6 Semantic Mapping

The purpose of defining a semantic mapping is to relate the syntactic concepts with the concepts in the semantic domain. Having defined the RTUML abstract syntax and well-formedness rules, and the semantic domain, we proceed with the definition of the semantic mapping. We first list the RTUML syntactic constructs and the concepts in the semantic domain, and then define the mapping by providing a semantic domain construct for each syntactic construct.

5.6.1 RTUML Syntactic Constructs

We list the syntactic constructs corresponding to each model element in the abstract syntax of RTUML.

- Class Diagram
  - Classifier Constructs
    - GRC Class (stereotype of Class) – aggregation of PortType classes and DataTypes
    - PortType Class (stereotype of Class) – associated set of input/output events
    - DataType – trait for abstract data type specification
    - Binding – instance of DataType parameterized with PortType classes/DataTypes
    - Attribute – instance of PortType class/DataType/DataType binding

- Association Constructs
    - PortAggregation Association (stereotype of Association) – binary relation between a GRC class and a PortType class
    - PortLink Association (stereotype of Association) – binary relation between two PortType classes

- Interaction Diagrams
  - Collaboration Diagram
    - Collaboration – instances of GRC/PortType classes and binary links between the instances
    - Classifier Role – instance of a GRC/PortType class
    - Association Role – instance of a PortAggregation/PortLink association
  - Sequence Diagram
    - Interaction – set of messages between instances of GRC classes
    - Message – from sender to receiver instances of GRC classes

- Statechart Diagram
  - State Machine – set of transitions between states
  - Transition – from source state to target state
  - State – simple or composite
  - Simple State
  - Composite State – submachine
  - Event – signal event triggering a transition
  - Guard – condition for a transition
  - Action – effect of a transition

5.6.2 Semantic Domain Concepts

We list the concepts of the semantic domain, that provide a definition for our notion of reactive object and system models.

- Time

- Types
  - GRCType – structure and behavior of a class of reactive objects
  - PortType – abstraction for events allowed at an instance of the port type
  - DataType – abstract data structure

- Event

- Internal event – assumed to occur at the *null port*
- External event – corresponding input and output events occurring at a port
- Input event – stimulus from another reactive object
- Output event – message sent to another reactive object (often as a reaction to a stimulus)

- State
  - Simple state
  - Complex state – with substates and transitions
  - Substates – mapping from a complex state to a set of states
  - Entry state – mapping from a complex state to a simple state
- Transition
  - Source state – prior to transition
  - Destination state – following transition
  - Trigger event – causing transition
  - Guard – enabling condition, port condition and timing constraint
  - Action – post condition and clock initialization for time-constrained reaction
- GRCType
  - GRC Class
    * Set of PortTypes – owned by the GRC class
    * Set of Attributes – part of the GRC class
    * Mapping from a PortType to a set of event names
  - ExtendedStatechart
    * Set of States
    * Set of Clocks – used in specifying time-constrained reactions
    * Set of Transitions
    * Mapping from a Clock to a Transition where the clock is initialized
    * Mapping from a Transition to a set of Clocks specifying the timing constraint
    * Mapping from a Clock to a set of States – disabling states for the timing constraint
    * Mapping from a Clock and a Transition to the Time at which the transition occurs
- Subsystem
  - Structure
    * Configuration
      · Set of reactive objects – instances of GRCTypes

        - Set of port objects – instances of PortTypes
        - Mapping from a reactive object to a set of port objects – port ownership
        - Mapping from a port object to another port object – communication channel
  - Communication
    - Scenario
        - Set of reactive objects – instances of GRCTypes
        - Sequence of messages – between instances of GRCTypes
    - Message
        - Sender object – instance of GRCType
        - Receiver object – instance of GRCType
        - Event labeling message
        - Occurrence time

5.6.3 Mapping Model Elements to Semantic Domain Concepts

In order to provide a complete mapping of RTUML syntactic constructs to concepts in the semantic domain, we associate each syntactic construct with a semantic domain concept, and provide informal characterizations of the associations. Figure 33 illustrates the mapping from RTUML syntax to concepts in the semantic domain: Table 4 shows how each diagrammatic component of RTUML is mapped onto a concept in the semantic domain; and Table 5 gives the mapping from each model element to a semantic domain concept. Appendix E provides the PVS specification of the semantic mapping.

RTUML Diagram	Semantic Domain
Class diagram	Types
Statechart diagram	Extended statechart
Collaboration diagram	Configuration
Sequence diagram	Scenario

Table 4: Semantic Domains for RTUML Diagrams.

- GRC Class model element to *GRC Class* semantic domain concept

  1. The number of Attributes of a GRC Class model element is equal to the sum of the number of *port attributes* and the number of *data attributes* of the corresponding *GRC Class* in the semantic domain.
  2. The number of Attributes whose types are PortTypes in a GRC Class model element is equal to the number of *port attributes* of the corresponding *GRC Class* in the semantic domain.
  3. The number of Attributes whose types are DataTypes or Bindings in a GRC Class model element is equal to the number of *data attributes* of the corresponding *GRC Class* in the semantic domain.

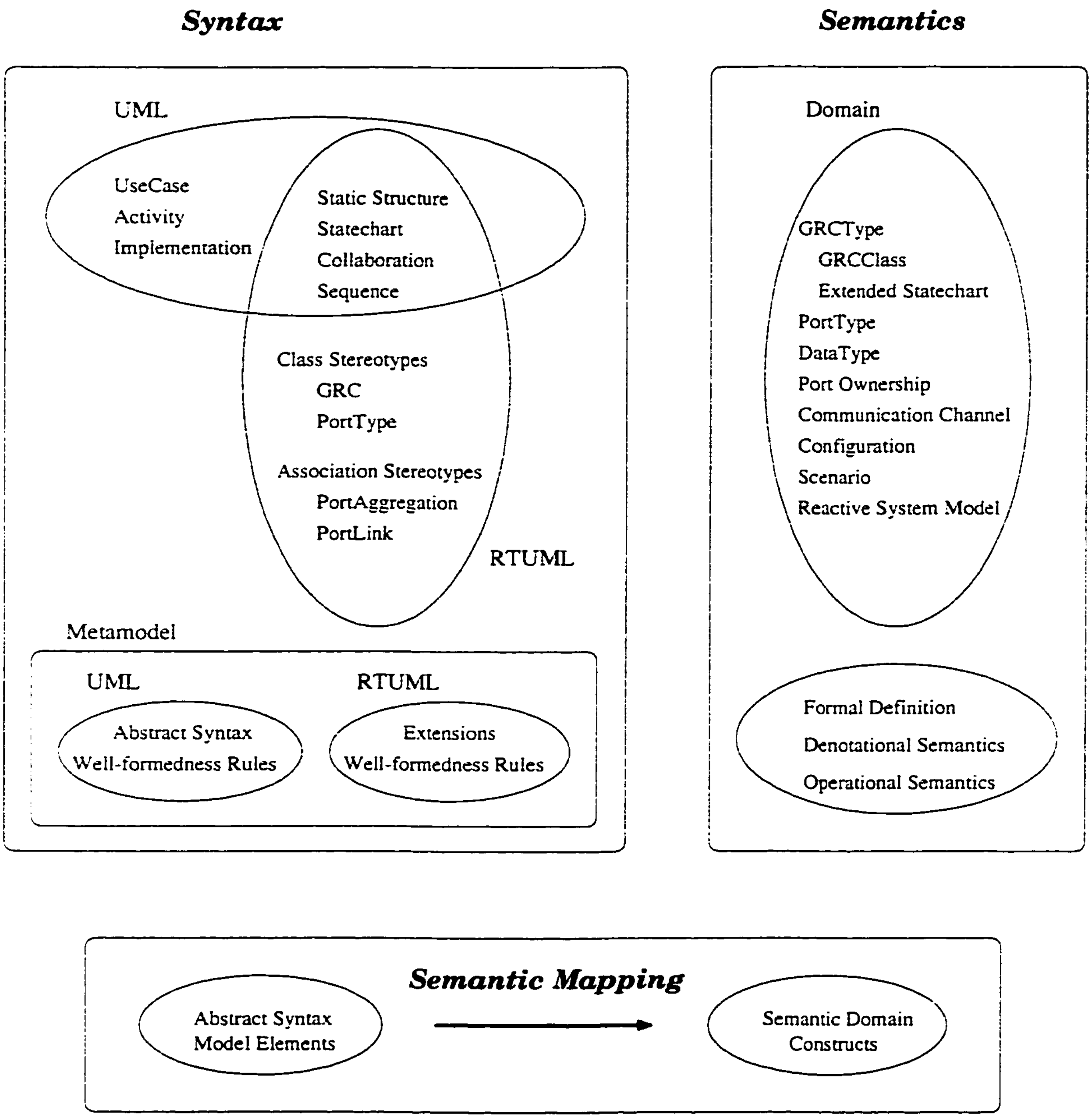


Figure 33: Semantic Mapping for RTUML.

	RTUML Model Element	Semantic Domain Concept
1	GRC Class	GRC Class
2	PortType Class	PortType
3	DataType	DataType
4	Binding	DataType
5	Attribute	PortTypeAttribute of a GRC Class
6	Attribute	DataTypeAttribute of a GRC Class
7	Attribute	Events *allowed* for a PortType
8	PortAggregation Association	PortType Ownership
9	PortLink Association	Communication Channel (*compatible* PortTypes)
10	Collaboration	Configuration
11	Classifier Role	Reactive Object
12	Classifier Role	Port Object
13	Association Role	Port Ownership
14	Association Role	Communication Channel (between 2 Port Objects)
15	Interaction	Scenario
16	Message	Message
17	State Machine	Extended Statechart
18	Transition	Transition
19	State	State
20	Simple State	Simple State
21	Composite State	Complex State
22	Event	Event
23	Guard	Guard
24	Action	Action
25	GRC Class + State Machine	GRCType

Table 5: Semantic Mapping from Model Elements to Semantic Domain Concepts.

4. Every Attribute whose type is a PortType in a GRC Class model element maps to a *port attribute* of the corresponding *GRC Class* in the semantic domain, and every Attribute whose type is either a DataType or a Binding in a GRC Class model element maps to a *data attribute* of the corresponding *GRC Class* in the semantic domain.
5. The number of AssociationEnds of a GRC Class model element is equal to the number of *PortTypes* of the corresponding *GRC Class* in the semantic domain.
6. For every opposite AssociationEnd of a GRC Class model element, there is a PortType Class model element that maps to a *PortType* that is owned by the corresponding *GRC Class* in the semantic domain.

- PortType Class model element to *PortType* semantic domain concept
    1. For every opposite AssociationEnd of a PortType Class model element, if the type of the AssociationEnd is a GRC Class model element, then, in the semantic domain, the corresponding *PortType* is owned by the corresponding *GRC Class*.
    2. For every opposite AssociationEnd of a PortType Class model element, if the type of the AssociationEnd is another PortType Class model element, then the PortType Classes map to

two *compatible PortTypes* in the semantic domain.

3. The Attribute of a PortType Class model element maps to the set of *events allowed* at the corresponding *PortType* in the semantic domain.

- PortAggregation Association model element to *PortType Ownership* semantic domain concept

  1. A PortAggregation Association model element maps to a tuple consisting of a *GRC Class* and a *PortType* in the semantic domain, such that the *GRC Class* owns the *PortType*.

- PortLink Association model element to *Communication Channel* semantic domain concept

  1. A PortLink Association model element maps to a tuple consisting of two *PortTypes* in the semantic domain, such that the two *PortTypes* are *compatible*.

- Collaboration model element to *Configuration* semantic domain concept

  1. A Collaboration model element maps to a *configuration* in the semantic domain, such that the number of ClassifierRoles in the Collaboration is equal to the sum of the number of *reactive objects* and the number of *port objects* in the corresponding *configuration*.
  2. If a Collaboration model element maps to a *configuration* in the semantic domain, then every ClassifierRole in the Collaboration maps to either a *reactive object* or a *port object* in the corresponding *configuration*.
  3. If a Collaboration model element maps to a *configuration* in the semantic domain, then every AssociationRole in the Collaboration maps to either a *port ownership* relation between a reactive object and a port object, or a *communication channel* between two port objects in the corresponding *configuration*.

- Interaction model element to *Scenario* semantic domain concept

  1. An Interaction model element maps to a *scenario* in the semantic domain, such that the number of Messages in the Interaction is equal to the number of *messages* in the *scenario*.
  2. If an Interaction model element maps to a *scenario* in the semantic domain, then every Message in the Interaction maps to a *message* in the *scenario*.
  3. If an Interaction model element maps to a *scenario* in the semantic domain, and a Message m_1 in the Interaction maps to a *message* m_2 in the *scenario*, then the sender of Message m_1 maps to the *sender* of *message* m_2, and the receiver of Message m_1 maps to the *receiver* of *message* m_2.

- State Machine model element to *Extended Statechart* semantic domain concept

  1. The top State of a State Machine model element maps to the *initial state* of the corresponding *extended statechart* in the semantic domain.
  2. The number of Transitions in a State Machine model element is equal to the number of *transitions* in the corresponding *extended statechart* in the semantic domain.

3. Every Transition in a State Machine model element maps to a *transition* in the corresponding *extended statechart* in the semantic domain.
4. If Transition t_1 in a State Machine model element maps to *transition* t_2 in the corresponding *extended statechart* in the semantic domain, then the *source* State of Transition t_1 maps to the *source* state of transition t_2, and the *target* State of Transition t_1 maps to the *destination* state of *transition* t_2.
5. If Transition t_1 in a State Machine model element maps to *transition* t_2 in the corresponding *extended statechart* in the semantic domain, and the *source* State of Transition t_1 is a SimpleState, then the corresponding *source* state of transition t_2 is a *simple* state. If the *target* State of Transition t_1 is a SimpleState, then the corresponding *destination* state of *transition* t_2 is a *simple* state.
6. If Transition t_1 in a State Machine model element maps to *transition* t_2 in the corresponding *extended statechart* in the semantic domain, and the *source* State of Transition t_1 is a CompositeState, then the corresponding *source* state of transition t_2 is a *complex* state. If the *target* State of Transition t_1 is a CompositeState, then the corresponding *destination* state of *transition* t_2 is a *complex* state.
7. If Transition t_1 in a State Machine model element maps to *transition* t_2 in the corresponding *extended statechart* in the semantic domain, then the trigger Event of Transition t_1 maps to the *trigger* event of *transition* t_2.
8. If Transition t_1 in a State Machine model element maps to *transition* t_2 in the corresponding *extended statechart* in the semantic domain, then the guard of Transition t_1 maps to the *guard* of *transition* t_2.
9. If Transition t_1 in a State Machine model element maps to *transition* t_2 in the corresponding *extended statechart* in the semantic domain, then the effect Action of Transition t_1 maps to the *action* of *transition* t_2.

- Composite State model element to *complex State* semantic domain concept

  1. The number of subvertices in a Composite State model element is equal to the number of *substates* in the corresponding *complex state* in the semantic domain.
  2. Every subvertex in a Composite State model element maps to a *substate* in the corresponding *complex state* in the semantic domain.

- GRC Class + State Machine model elements to *GRCType* semantic domain concept

  1. If a tuple consisting of a GRC Class and a StateMachine model elements map to a *GRCType* in the semantic domain concept, then the GRC Class model element maps to the *GRC Class* that is part of the *GRCType*, and the StateMachine model element maps to the *extended statechart* that is part of the *GRCType*.

5.7 Operational Semantics

A logical semantics for the abstract reactive model serves two purposes: (i) as a set of rules for checking the well-formedness of UML models of real-time reactive systems, and (ii) as a foundation for a formal verification methodology. A logical semantics suitable for analysis and reasoning corresponds to an axiomatization [Ach95] of the structural and behavioral characteristics of the abstract reactive model. To obtain the axiomatic description of a reactive object, we substitute the formal arguments of predicates in the axioms from the logical semantics with data from the status of the object. The resulting set of axioms for the objects, together with synchronization axioms describing communication protocols within a subsystem configuration, supports a rigorous analysis of the design. Appendix F gives the PVS specification of the RTUML operational semantics.

Our goal is to provide a semantic framework as a basis for the specification and analysis techniques. Since UML metamodel and well-formedness rules are defined in OCL, we endeavor to define formal semantics for the abstract model in OCL. Although OCL is convenient for describing modeling elements of object-oriented notations, it is not tailored for real-time specifications. To support a semantic definition of the structure and behavior of real-time reactive objects and subsystems specified according to the proposed modeling technique, we need a notation that allows the definition of logical assertions, constraints, and properties involving time. Incorporating temporal predicates within the first-order logical framework of OCL, we provide a semantic basis for UML design specifications based on the abstract real-time reactive model.

We introduce a reactive object domain in OCL to give semantic definitions that apply to instances of the generic reactive classes. We define an abstract base class *GRC* as a generalization of the generic classes GRC_i. The class *GRC* encapsulates the following data: *port types*, *states*, *events*, *attributes*, *transition specifications*, and *time constraints*. A class GRC_i defines a set of *ports* for each of its *port types*, and an *attribute function* giving the active attributes in each state of instances of the reactive class. This definition allows access to data relevant to instances of the reactive classes.

To provide semantics for subsystem configurations, we introduce a reactive subsystem domain in OCL. We view a collaboration diagram as a configuration of instances of generic reactive classes. We define an abstract base class *Subsystem* as a generalization of the subsystem configuration specifications. This class represents a composition of *instances* and *portlinks*; the attribute *instances* defines a set of objects of the generic reactive classes, and the attribute *portlinks* defines a set of binary associations between ports of the reactive objects. A tuple of the form

$$<< instance_a,\ port_m >,\ < instance_b,\ port_n >>$$

defines a port-link for a communication channel between port $port_m$ of object $instance_a$ and port $port_n$ of object $instance_b$. In this model, the only form of communication between reactive objects is through synchronous message passing.

We introduce a domain for time intervals in OCL, and define the temporal predicates shown in Figure 34 to express relations on time intervals. The equality predicate *Equal* is symmetric; the predicates *Before*, *Meet*, *Overlaps*, *During*, *Starts*, and *Finishes* are asymmetric. Thus, for a pair of time intervals, there are thirteen relations expressible by these predicates and their inverses. More complex temporal relations on time intervals can be expressed in terms of these predicates. Table 6 gives the definitions of the temporal predicates

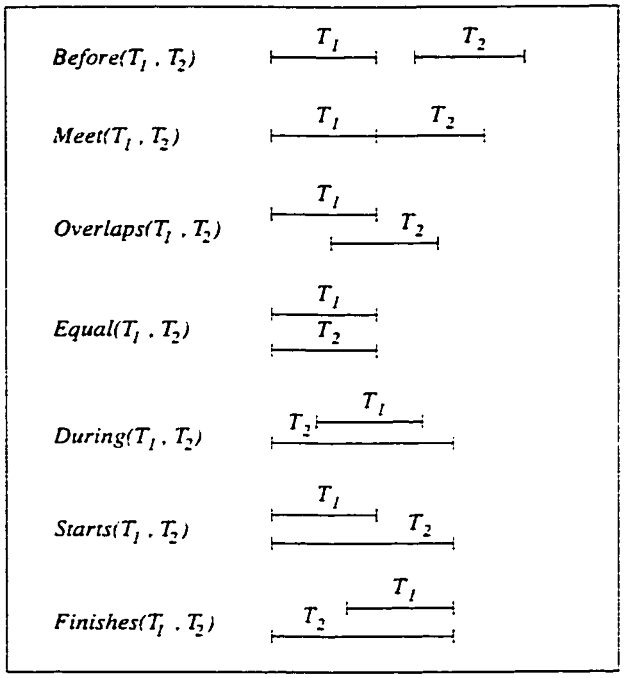


Figure 34: Temporal Predicates.

$Before(T_1, T_2)$	$\triangleq$	$v_1 < u_2$
$Meet(T_1, T_2)$	$\triangleq$	$v_1 = u_2$
$Overlaps(T_1, T_2)$	$\triangleq$	$u_1 < u_2 < v_1 < v_2$
$Equal(T_1, T_2)$	$\triangleq$	$u_1 = u_2 \wedge v_1 = v_2$
$During(T_1, T_2)$	$\triangleq$	$u_2 < u_1 \wedge v_1 < v_2$
$Starts(T_1, T_2)$	$\triangleq$	$u_1 = u_2 \wedge v_1 < v_2$
$Finishes(T_1, T_2)$	$\triangleq$	$u_1 < u_2 \wedge v_1 = v_2$

Table 6: Definitions of the Temporal Predicates.

on time intervals. The variables T_1 and T_2 range over the domain of time intervals; $T_1 = [u_1, v_1], v_1 > u_1$, and $T_2 = [u_2, v_2], v_2 > u_2$. For every attribute x of an object, we define a function x whose domain is the set of time intervals, such that for every member of the set, the object remains in the same state during the interval. The function returns the value of the attribute in the given interval.

We define predicates on time intervals to assert time-dependent properties on elements from the domain of reactive objects. To assert that a property holds for an object at time t, or during time interval T, we define the predicates *HoldAt* and *HoldDuring* to apply to reactive objects. The arguments of predicate *HoldAt* are a state s and a time point t; *HoldAt(s,t)* asserts that the object is in state s at time t. Similarly, predicate *HoldDuring(s,T)* asserts that the object is in state s during the time interval T. If time interval T = [u,v], v > u, then

$A.HoldDuring(s, T)$ *implies*

$\forall t : u \leq t \ and \ t \leq v \ implies \ A.HoldAt(s,t),$

denoting that if an object A is in state s during interval T, then object A is in state s at every time point t within the lower and upper bounds u and v of the interval. We define predicate *Occur* to assert the occurrence of an event at a port, at a specified time in a reactive object. The arguments of *Occur* are an event e, a port p, and the time t at which the event occurs. The abstract base class *GRC* encapsulates the predicates *HoldAt*, *HoldDuring* and *Occur*.

5.7.1 Axiom System in OCL

There are eleven axioms of temporal constraints associated with an instance of a generic reactive class. The *Synchrony* axiom describes the semantics for synchronous message passing. An OCL expression of the form

$$\text{self}.events\text{->forall}(e \mid P(e))$$

applied to a reactive object, denotes "*for all events e of the GRC instance, predicate P is true*". The variable t_i denotes the absolute time for an event occurrence; the variable s_i denotes a state in the automaton; variables e, f, and e_i denote events labeling transitions in the automaton; the variable p_i denotes a port of the reactive object. The predicate $HoldAt(s,t)$ asserts that the object is in state s at time t; the predicate $Occur(e,p,t)$ asserts that event e occurs at time t through port p in the reactive object.

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

   At time t, there can be at most one event occurring in a reactive object; at time t, an event can occur at only one port.

   (a) self.$events$->forall($e_1, e_2 \mid e_1 \neq e_2$
   and self.$ports$->exists($p_i \mid$ self.$Occur(e_1, p_i, t)$)
   implies self.$ports$->forall($p_j \mid$
   not self.$Occur(e_2, p_j, t)$)))

   (b) self.$events$->forall($e \mid$ self.$ports$->forall($p_i, p_j \mid$
   self.$Occur(e, p_i, t)$ and self.$Occur(e, p_j, t)$
   implies $p_i = p_j$))

2. *Silent-event axiom:* .... (SE)

   The occurrence of the silent event *tick* at time t precludes the occurrence of any other event in the reactive object at time t.

   self.$Occur(tick, nullport, t)$ implies
   self.$events$->forall($e \mid$ self.$ports$->forall($p_i \mid$
   not self.$Occur(e, p_i, t)$))

3. *State-hierarchy axioms:* .... (SH)

These axioms assert the relationship between a state and its substates. When an object is in a substate of a state θ, it is also in the state θ. Similarly, when a reactive object is in a non-atomic state θ, it is in at least one of the substates of θ.

(a) self.$states$->forall(s_j | $s_j.complex$ = true
  and $s_j.substates$->exists(s_i | self.$HoldDuring(s_i, T)$)
   implies self.$HoldDuring(s_j, T)$)

(b) self.$HoldDuring(s_j, T)$ and $s_j.complex$ = true
  implies $s_j.substates$->exists(s_i | self.$HoldDuring(s_i, T)$)

4. *State-uniqueness axiom:* .... (SU)

A reactive object cannot be in more than one state at any instant, unless the states are related by the state hierarchy function Φ_s. That is, a reactive object can be in two states only if one state is a substate of the other. Formally,

self.$HoldAt(s_1, t)$ and $s_1 \neq s_2$
 and not $s_2.substates$->exists(s_3 | $s_3 = s_1$)
  and not $s_1.substates$->exists(s_3 | $s_3 = s_2$)
   implies not self.$HoldAt(s_2, t)$

5. *Initial-state axiom:* .... (IS)

A reactive object has a unique initial state which is atomic. A reactive object is in its initial state θ_0 at the initial instant t_{init}.

self.$HoldAt(s, t_{init})$ implies
 $s.initial$ = true and $s.complex$ = false

6. *Initial-attribute axiom:* .... (IA)

A formula φ_{init} is asserted at the initial time t_{init} such that φ_{init} is the maximal property satisfied by the attributes at t_{init}. The assertion φ_{init} is the maximal property in the sense that, for any other assertion φ satisfied by the attributes at time t_{init}, the following holds:

$\varphi_{init}(t_{init})$ implies $\varphi(t_{init})$

7. *Dormant-attribute axiom:* .... (DA)

The attribute function partitions the attribute set into modifiable and non-modifiable sets, at each state. If an attribute is dormant in a certain state then its value cannot be changed as long as the machine is in that state.

self.$states$->forall(s |
  self.$HoldDuring(s, T_1)$ and $Meet(T_2, T_1)$
    implies self.$attributes$->forall(a |
      self.$attributefunc(s)$->forall(x_i |
        $x_i \neq a$ implies $x_i(T_2) = x_i(T_1)$))))

8. *Occurrence axiom:* .... (OC)

For the occurrence of signal $Occur(e, p_i, t)$ it is necessary that the reactive object be in the source-state of some transition λ, labeled by e, such that the port-condition φ_{port} of λ is satisfied by p_i. This is formalized by the occurrence axiom asserted for each event e in the reactive object. For an event e, let $\lambda_1, \ldots, \lambda_n$ be the transition specifications labeled by e, and let θ_j be the source-state of λ_j, φ^j_{en} be the enabling-condition of λ_j and φ^j_{port} be the port-condition of λ_j. The occurrence axiom for e follows.

self.$Occur(e, p_i, t)$
  implies self.$states$->exists(s | self.$HoldAt(s, t)$
    and self.$transitions$->exists(r | $r.source = s$
      and $r.enablingcondition(t) =$ true
        and $r.portcondition(t, p_i) =$ true))

9. *Transition axiom:* .... (TR)

The transition axiom is defined for each transition specification of a reactive object. The occurrence of an event results in a state transition to the target state and the satisfaction of the post-condition in the target state. The transition axiom applies for each transition specification $\lambda: \langle\theta, \theta'\rangle; e(\varphi_{port}): \varphi_{en} \Longrightarrow \varphi_{post}$. If the target state θ' is not an atomic state, then the atomic state which is the starting descendant state of θ' replaces θ'.

self.$HoldAt(s_1, t_1)$ and self.$Occur(e, p, t_1)$
  and $t_1 < t_2$ implies
    self.$transitions$->exists(r | $r.source = s_1$
      and $r.destination = s_2$ and self.$HoldAt(s_2, t_2)$
        and $r.postcondition(t_1, t_2, p) =$ true)

10. *Persistence axiom:* .... (PS)

A persistent axiom is defined for each state. It asserts that when no event causing a transition to leave that state occurs, there is neither a change in that state nor a change in the values of the attributes active in that state. For a state θ, let $e_1, \ldots, e_n$ denote the events associated with the transitions leaving θ. The persistent axiom for θ is as follows.

self.$HoldDuring(s, T_1)$ and $Meet(T_1, T_2)$
  and self.$transitions$->forall(r | $r.source = s$
    implies not self.$Occur(r.triggerevent, p_1, t_1)$))
      implies self.$HoldDuring(s, T_2)$
        and self.$attributefunction(s)$->forall(x |
          $x(T_1) = x(T_2)$))

11. *Time-constraint axioms:* .... (TC)

A set of time constraint axioms defines the behavior of a reactive object. The axioms apply for each time-constraint

$$(\lambda, e, [l, u], \Theta_i) = \upsilon_i \in \Upsilon,$$

where $\lambda : \langle\theta, \theta'\rangle; f(\varphi_{port}); \varphi_{en} \longrightarrow \varphi_{port}$. We introduce the predicates *Enable*, *Disable*, and *Trigger*, to describe the status of a reaction after it has been enabled, and the predicate *Within* to assert the containment of a time point within a bounded time interval.

- $Trigger(e, t_a)$: A reaction is activated when a transition triggering the reaction occurs. For a time-constraint υ_i, the occurrence of a trigger transition λ is marked by a change of state from θ to θ' and the occurrence of the labeling event f. $Trigger(e, t_a)$ is true when a reaction associated with the constrained event e is activated at time t_a. If e is not a constrained event then $\forall t,\ \neg Trigger(e, t)$ is true.

$Trigger(e, t_a) \stackrel{\text{def}}{=}$ self.$transitions$->exists(r |
  $r.triggerevent = f$ and self.$Occur(f, p_i, t_a)$
    and self.$HoldAt(r.source, t_1)$
      and self.$HoldAt(r.destination, t_2)$
        and $t_1 < t_a$ and $t_a < t_2$
          and self.$timeconstraints$->exists(tc |
            $tc.assoctransition = r$
              and $tc.constrainedevent = e$))

- $Disable(e,t)$: Any activated reaction involving the constrained event e is disabled at time t due to the reactive object entering one of the disabling states of e. If e is not a constrained event then $\forall t$, $\neg Disable(e,t)$ is true.

$$\begin{array}{l} Disable(e,t) \stackrel{\text{def}}{=} \text{self}.timeconstraints{-}{>}\text{exists}(tc \mid \\ \quad tc.constrainedevent = e \ \text{ and} \\ \quad\quad t < tc.upperbound \\ \quad\quad\quad \text{and } tc.disablingstates{-}{>}\text{exists}(s \mid \text{self}.HoldAt(s,t))) \end{array}$$

- $Enable(e,t_a,t)$: The reaction involving the constrained event e due to the occurrence of a trigger event at the activation instance t_a is enabled at time t. An event e is enabled at time t if it was triggered at time t_a, $t_a < t$, and it was not disabled or fired at any time t', $t_a < t' \leq t$. A formal definition of the predicate follows from the axioms stated below. If e is not a constrained event then $\forall t_a, t$, $\neg Enable(e,t_a,t)$ is true.
- We define the predicate $Within(t_a,l,u,t)$ in terms of the basic temporal predicates.

$$Within(t_a,l,u,t) \stackrel{\text{def}}{=} t_a + l \leq t \leq t_a + u$$

The following axioms use the predicates $Trigger(e,t_a)$, $Disable(e,t)$, and $Enable(e,t_a,t)$, and the temporal predicates to describe the behavior of objects of the generic reactive classes.

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

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

$$\begin{array}{l} \text{self}.timeconstraints{-}{>}\text{forall}(tc \mid tc.constrainedevent = e \\ \quad \text{and } Trigger(e,t_a) \text{ and not } Disable(e,t) \\ \quad\quad \text{and } t_a < t \text{ implies } Enable(e,t_a,t)) \end{array}$$

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

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

$$\begin{array}{l} \text{self}.timeconstraints{-}{>}\text{forall}(tc \mid tc.constrainedevent = e_1 \\ \quad \text{and self}.Occur(e_1,p_i,t) \text{ implies} \\ \quad\quad \text{self}.transitions{-}{>}\text{exists}(r \mid r.triggerevent = e_2 \\ \quad\quad\quad \text{and self}.Occur(e_2,p_j,t_a) \\ \quad\quad\quad\quad \text{and } t_a + l \leq t \text{ and } t \leq t_a + u)) \end{array}$$

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

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

self.$timeconstraints$->forall(tc | $tc.constrainedevent = e$
  and $Enable(e, t_a, t)$ and not self.$Occur(e, p_i, t)$
    and $t < t_2$ and not $Disable(e, t_2)$
      implies $Enable(e, t_a, t_2)$)

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

An enabled reaction will no longer be enabled if the constrained event of the reaction is disabled due to the object entering into a disabling state.

self.$timeconstraints$->forall(tc | $tc.constrainedevent = e$
  and $Enable(e, t_a, t)$ and $t < t_2$
    and $Disable(e, t_2)$ implies not $Enable(e, t_a, t_2)$)

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

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 be enabled.

self.$timeconstraints$->forall(tc | $tc.constrainedevent = e$
  and $Enable(e, t_a, t)$ and self.$Occur(e, p_i, t)$
    and $Within(t_a, 0, u, t)$ and $t < t_2$
      implies not $Enable(e, t_a, t_2)$)

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

If a reaction is enabled then the constrained event should not occur during the minimum delay period from the time of activation. However, if the minimum delay is less than the atomic interval, then there does not exist any minimum delay interval.

self.$timeconstraints$->forall(tc | $tc.constrainedevent = e$
  and $Enable(e, t_a, t)$ and $Within(t_a, 0, l, t)$
    implies not self.$Occur(e, p_i, t)$)

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

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

$\text{self}.\mathit{timeconstraints}\text{->forall}(tc \mid tc.\mathit{constrainedevent} = e$
$\quad \text{and } Enable(e,t_a,t) \text{ and}$
$\quad\quad (Within(t_a,0,u,t_2) \text{ implies not } Disable(e,t_2))$
$\quad\quad\quad \text{implies self.}Occur(e,p_i,t_2) \text{ and } Within(t_a,l,u,t_2))$

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

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 bound u from the instant t_a. In other words, for a constrained event activated at a given time t_a, for all time instants t', such that, $t' < t_a$ or $t' > t_a + u$, the constrained event e cannot be enabled. By including this axiom, we can assert whether or not the predicate $Enable(e,t_a,t)$ is true for all constrained events e and time instants t_a and t.

$\text{self}.\mathit{timeconstraints}\text{->forall}(tc \mid tc.\mathit{constrainedevent} = e$
$\quad \text{and } Enable(e,t_a,t)$
$\quad\quad \text{implies } Trigger(e,t_a) \text{ and } Within(t_a,0,u,t))$

12. *Synchrony axiom:* .... (SY)

The synchrony axiom applies for each port-link $o_i.@q_j \leftrightarrow o_k.@q_l$ in a *Subsystem Configuration Specification.*

$\text{self}.\mathit{portlinks}\text{->forall}(pl \mid \text{self}.\mathit{instances}\text{->exists}(o_1,o_2 \mid$
$\quad pl.instance_1 = o_1 \text{ and } pl.instance_2 = o_2$
$\quad\quad \text{and } (o_1.Occur(e,pl.port_1,t)$
$\quad\quad\quad \text{implies } o_2.Occur(e,pl.port_2,t))$
$\quad\quad \text{and } (o_2.Occur(e,pl.port_2,t)$
$\quad\quad\quad \text{implies } o_1.Occur(e,pl.port_1,t)))$

Chapter 6

Mechanized Verification Methodology

Proving safety, liveness, and bounded-time properties is crucial in the development of dependable safety-critical systems. Attaining this goal at an early design stage is essential in reducing the duration and cost of the development life cycle. We present a methodology for mechanized verification of time-dependent properties of real-time reactive systems. The verification technique is based on an object-oriented formal specification method. The methodology applies to properties that can be specified in terms of relations on absolute times for event occurrences. This technique is pertinent to object-oriented formalisms based on state machines augmented with timing constraints and synchronous communication.

6.1 Introduction

This chapter is organized as follows. Section 6.2 introduces the proposed verification methodology and its underlying principles. It presents a method for deriving an axiomatic description from formal design specifications, and outlines the verification methodology for proving time-dependent properties in a design. Section 6.3 illustrates the verification methodology with the safety property of the generalized railroad crossing problem. Section 6.4 describes the operational semantics as a logical foundation for the verification methodology. Section 6.5 makes use of the railroad crossing problem to illustrate the equivalence between the two sets of axioms, and outlines the blueprint for an automated mechanism for deriving the axioms needed in the verification process.

6.2 Formal Verification

The desired properties of a reactive system are usually not expressible in terms of the behavior of the unit alone; instead, they are statements about interaction between the unit and its environment. To guarantee an acceptable behavior of the unit, environmental properties on which the unit can rely have to be ascertained. The desired properties become theorems to be proved from the dynamic model of the system. A program representing an arbitrary behavior of the design will satisfy the properties. When a certain property cannot be proved, we redefine system requirements, and redesign the system.

The basis for the underlying object-oriented formalism is an augmented state machine modeling time-constrained reactive object behavior. With encapsulation of transition specifications and time constraints in generic reactive classes, the behavior of every instance of a class conforms to the same specifications. A formal approach to analysis of design specifications and reasoning about time-dependent behavior becomes possible prior to implementation. The proposed verification methodology relies on the object-oriented nature of the model for dealing with complex systems involving numerous objects and intense interaction. A behavioral semantics for reactive systems specified in conformance with this modeling technique provides a sound basis for formal verification.

The methodology applies to verification of time-dependent properties, such as safety and liveness requirements. Using UML graphical notation as a front-end in the design environment makes the underlying formal notation transparent to the user. The significance of this approach lies in the integration of PVS as a back-end for mechanized verification of systems described in UML. The foundations of the methodology rest on the behavioral semantics of the abstract object-oriented model for real-time reactive systems [AAM96]. We automate the methodology within PVS environment by providing a translator to derive axioms from formal design specifications, and using the interactive theorem prover for reasoning about the time-constrained behavior of reactive systems. The verification methodology is applicable to object-oriented formalisms based on state automata augmented with time constraints.

6.2.1 Axiomatic Description of Design Specifications

We introduce the following concepts for the axiomatic description of design specifications. The *computation* of an object consists of a sequence of state transitions starting from the *initial* state of the object, and corresponds to a time-series of event occurrences. Since a reactive system is in continuous interaction with its environment, the sequence can be infinite. A *period* corresponds to a segment of the sequence of state transitions in the computation of an object, whereby the object is in its initial state both at the beginning and at the end of the segment, and the object is not in the initial state in-between. Since the periodicity applies to functionality and not to time, successive periods in the computation of an object need not have the same execution time. Moreover, a periodic task does not entail the same sequence of event occurrences in each period. Since events are recurrent, a period can include multiple *occurrences* of an event. *Transition times* correspond to absolute times for occurrences of events within a period; each event occurrence corresponds to a transition in the underlying state machine for the reactive object.

For each generic reactive class *GRC*, we define a higher-order function *TT* that gives the transition time corresponding to an occurrence of an event within a certain period for an instance of the class. The signature of the function conforms to the following schema.

$$TT : [GRC \text{ −> } [Period \text{ −> } [GRC\_Event \text{ −> } [Occurrence \text{ −> } Time]]]].$$

GRC is the set of all instances of the generic reactive class, *Period* is the set of all periods in the computation of the object, *GRC_Event* is the set of events for the generic reactive class, *Occurrence* is the set of all occurrences of the event within the period, and *Time* is the set of all possible discrete time values. The sets *Period* and *Occurrence* may be finite or denumerably infinite, and are interpreted as the set of positive natural numbers; the set *Time* is denumerably infinite, and is interpreted as the set of non-negative real numbers. Defining the function *TT* in a parameterized theory with formal arguments for the type of objects and for the type of events in the objects allows us to instantiate the theory for each generic reactive class; the function is thus overloaded for each class.

For a given object A and an event e, the function *TT* is monotonic increasing with respect to event occurrence indices within a period, and monotonic increasing with respect to period indices. Therefore,

$$TT(A)(i)(e)(j+1) \; > \; TT(A)(i)(e)(j), \;\; \forall\, i,\, j,$$

and

$$TT(A)(i+1)(e)(j_1) \; > \; TT(A)(i)(e)(j_2), \;\; \forall\, i,\, j_1,\, j_2.$$

The function supports the specification of various relations on transition times, both within a specific period and across periods. For instance, within the i–th period, the j–th occurrence of event e_1 may precede the j–th occurrence of event e_2, and the j–th occurrence of event e_2 may precede the $(j+1)$–th occurrence of event e_1. Similarly, there can be a time constraint between the j–th occurrence of event e_1 in the i–th period and the k–th occurrence of event e_2 in the $(i+1)$–th period. In addition, relations on transition times may involve multiple objects from the system configuration.

From the formal specifications of a reactive system design, we derive axioms describing the time-dependent behavior of the system. We group them into four categories: *transition*, *time constraint*, *synchrony*, and *supplementary* axioms. The transition, time constraint, and synchrony axioms specify ordering relations on transitions, time constraints on reactions to a transition, and synchronization of message exchanges, respectively. The supplementary axioms describe other design aspects that capture specific requirements essential for a correct functional and temporal behavior of the system.

A generic reactive class encapsulates transition specifications; hence, a *transition* axiom is specific to a class. A *transition* axiom specifies an ordering relation on two valid transitions in the state machine for the class. Starting from the initial state, if the destination state of transition R_1 is the same as the source state of transition R_2, we include an axiom specifying that the transition time for the j–th occurrence of the trigger event of transition R_1 precedes the transition time for the j–th occurrence of the trigger event of transition R_2. These event occurrences must happen within the same period in the computation of the reactive object. Similarly, an occurrence of the trigger event of a transition from the initial state within the i–th period succeeds all occurrences of the trigger event of a transition leading to the initial state in the $(i-1)$–th period. In extracting the transition axioms, care must be exercised in dealing with transitions that lead to a state which has already been *visited* within the same period. For instance, a reflexive transition labeled with an event e can lead to several occurrences of the event e in the same period. Similarly, a transition from state s_1 to state s_2, such that state s_2 has already been visited in the same period and state s_2 is not the initial state, may cause several occurrences of an event e within that period. The ordering relation must be asserted on specific event occurrences. For an object A, and events e_1 and e_2, the axiom is expressed in the form

$$TT(A)(i)(e_1)(j) \; < \; TT(A)(i)(e_2)(j),$$

where an occurrence of event e_2 always follows an occurrence of event e_1.

A generic reactive class encapsulates timing constraints on reactions to transitions; a *time constraint* axiom is therefore specific to a class. In deriving *time constraint* axioms, we consider the time interval during which a reaction to a transition is constrained to occur. The occurrence of an event e_1, in reaction to the occurrence of an event e_2 at time t, is constrained to befall within the lower and upper time bounds l and u relative to time t. If the object is in the source state of the transition labeled with event e_1, then the time elapsed since the reaction was enabled is less than the upper bound u. On the other hand, if the object is in the destination state of the transition labeled with event e_1, then the time elapsed since the reaction was enabled is greater than the lower bound l. The axioms state that the *delay* between the occurrence of the *activation* event e_2 and the occurrence of the *reaction* event e_1 is greater than l and less than u. The axioms apply to event occurrences within the same period in the computation of the reactive object. For an object A, and events e_1 and e_2, the axiom describes a relation on event occurrence times, and is expressed in the form

$$TT(A)(i)(e_1)(j) \; > \; TT(A)(i)(e_2)(j) \, + \, l \;\text{ and}$$

$$TT(A)(i)(e_1)(j) \; < \; TT(A)(i)(e_2)(j) \, + \, u,$$

where the interval $[l, u]$ specifies the window relative to the occurrence time of event e_2, during which event e_1 can occur in reaction, and l and u are natural numbers such that $u \; \geq \; l$.

The *synchrony* axioms capture the message passing mechanism between instances of the generic reactive classes. Messages correspond to events specified as an *input* event in one class and as an *output* event in another class. A *synchrony* axiom is valid only for pairs of objects that communicate through a *port-link* defined in the subsystem configuration. We derive *synchrony* axioms from transition specifications corresponding to occurrences of *external* events. A synchronized message causes simultaneous transitions in two communicating objects. The axioms state that for every occurrence of an output event e at time t in an object A_1, there exists a corresponding occurrence of an input event e at time t in a *linked* object A_2, where objects A_1

and A_2 are instances of different reactive classes. For objects A_1 and A_2, and synchronized event e, the j–th occurrence of event e happens at the same time in both objects within the i–th period. The axiom describes a relation on event occurrence times, and is expressed in the form

$$TT(A_1)(i)(e)(j) \;=\; TT(A_2)(i)(e)(j).$$

We structure the PVS specifications in terms of theories. The theory *model* defines the *Time* domain as the set of non-negative real numbers. It also includes definitions for the *Period* and *Occurrence* domains as positive natural numbers. The parameterized theory *transition_time* defines the function *TT*. The formal arguments of the theory are the types *GRC* and *GRC_Event*, corresponding to the type of objects and to the type of events occurring in the objects. The appendices include the PVS theories *model* and *transition_time*. For each generic reactive class, we define a PVS theory containing the following:

- an uninterpreted nonempty type for the set of instances of the class,
- an enumerated type defining the set of events occurring in the objects of the class,
- an instantiation of the *transition_time* theory with the two types defined above, to overload the function *TT* for the class,
- declarations for universally quantified logical variables for *Period*, *Occurrence*, and the class of objects,
- a set of *transition* axioms, and
- a set of *time constraint* axioms.

For each subsystem configuration, we define a theory containing the following:

- an *importing* clause to include the theories defining the reactive classes,
- an *importing* clause to import the other subsystems included in the subsystem,
- declarations for universally quantified logical variables for *Period* and *Occurrence*,
- declarations for universally quantified logical variables for the classes of objects, for the case of a general proof,
- declarations for constants corresponding to the objects instantiated in the subsystem,
- a set of *synchrony* axioms, and
- a set of *supplementary* axioms.

The theory describing the overall system includes the theorem specifying the property to be proved. This structure provides a template for automating the generation of axiomatic descriptions from formal specifications. The generation of most transition, time constraint and synchrony axioms can be automated; however, this may not apply to the supplementary axioms.

6.3 Case Study – Generalized Railroad Crossing

We illustrate the verification methodology with the generalized railroad crossing problem [HL94], a benchmark for evaluating formal methods for real-time computing. Several trains, traveling in different directions, traverse a crossing independently and simultaneously using multiple non-overlapping tracks. A controller is responsible for the safe operation of the gate at a crossing. When a train traverses a crossing, it communicates with the relevant controller. The controller commands the gate to close when it receives a message from the first train entering the crossing. The controller instructs the gate to reopen when it receives a message from the last train leaving the crossing. The safety requirement stipulates that whenever there is a train inside a crossing, the corresponding gate remains closed.

Considering that there is one gate at each crossing, and that a unique controller maneuvers the gate, we observe that the behavior of the controllers are independent of each other. Similarly, since a gate is controlled by a single controller, we can assert that the behavior of a gate is independent of the behavior of the other gates. Since all objects of a generic reactive class have a similar structure and behavior, we can conclude that if one *controller-gate* pair operates safely, then all the other controller-gate pairs are safe. We therefore verify the safety property for each controller-gate pair individually. To maintain generality, we assume an arbitrary number of trains in the system, and consider that all trains interact with the controller.

In the following subsections, we specify and analyze the design of the railroad crossing system. We describe the UML diagrams modeling the reactive entities, and the formal specifications generated from the diagrams. We specify the *transition*, *time constraint*, *synchrony*, and *supplementary* axioms for the system with an arbitrary number of trains. We formulate the safety property for the system as a theorem, and construct a proof for the theorem within PVS verification environment. Finally, we establish the correctness of the proof for the safety property, using time sequence charts to describe the behavior of the system.

6.3.1 Formal Specifications

The safe operation of the control system depends on the satisfaction of several timing constraints, so that the gate is closed before a train enters the crossing, and the gate is opened after all trains leave the crossing.

- A train enters the crossing within an interval of 2 to 4 time units after informing the controller that it is approaching.
- A train informs the controller that it is leaving the crossing within 6 time units of sending the approaching message.
- The controller instructs the gate to close within 1 time unit after receiving the first approaching message, and starts monitoring the gate.
- The controller continues to monitor the closed gate when it receives an approaching message from other trains, and as long as there is a train inside the crossing.
- The controller instructs the gate to open within 1 time unit after receiving an exiting message from the last train leaving the crossing.
- The gate must close within 1 time unit of receiving instructions from the controller.

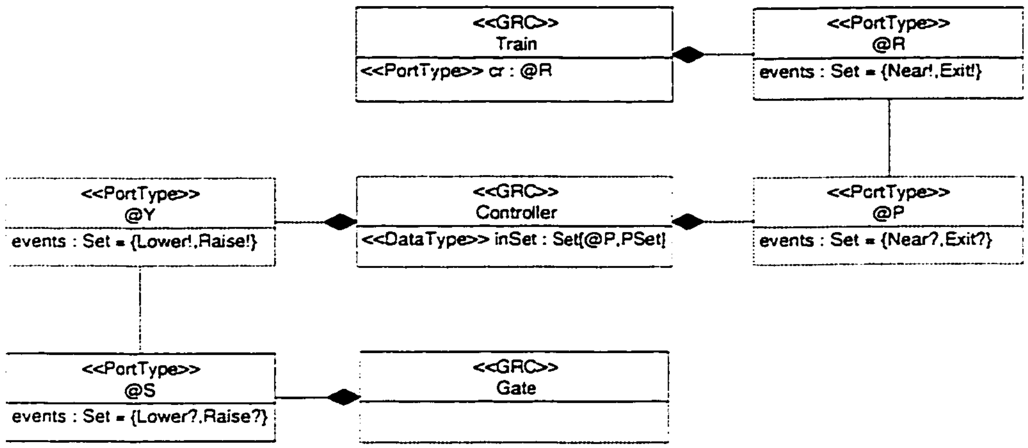


Figure 35: Class Diagram for Train, Controller, and Gate GRC's.

- The gate must open within an interval of 1 to 2 time units of receiving instructions from the controller.

Initially in the state *idle*, a train sends the event *Near* to the controller on approaching a crossing, and enters the state *toCross*. The internal events *In* and *Out* correspond to the train entering and leaving the crossing. The state *cross* corresponds to the train being inside the crossing, and the state *leave* corresponds to the train having exited the crossing and has yet to inform the controller. The train goes back to the state *idle* upon sending the message *Exit* to the controller.

The controller is initially in the state *idle*; it enters the state *activate* upon receiving the first *Near* message. The controller sends the event *Lower* to the gate it maneuvers, and enters the state *monitor*. Subsequent *Near* messages received by the controller in the state *activate* or *monitor* do not cause a state change. When the controller receives *Exit* messages from trains leaving the crossing, it stays in the state *monitor* until the last train leaves, at which time it enters the state *deactivate*. The controller then sends the event *Raise* to the gate, and goes back to the state *idle*. The attribute *inset* of the GRC class *Controller* corresponds to an instance of the LSL trait *Set*; it keeps track of the trains that are inside the crossing.

The initial state of the gate is *opened*; when it receives the message *Lower* from the controller, it enters the state *toClose*. The internal event *Down* corresponds to the closing of the gate, upon which the gate enters the state *closed*. On receiving the message *Raise*, the gate enters the state *toOpen*. The internal event *Up* corresponds to the re-opening of the gate; it then goes back to the state *opened*.

A controller detects trains approaching and leaving the crossing through sensors. It is reasonable to consider the events capturing these messages to be synchronized between the controller and the train. Similarly, the controller monitors the operations of the gate through actuators. The messages for lowering and raising the gate correspond to synchronized events between the controller and the gate. Since a controller monitors a gate as long as the gate is closed, and it is idle when the gate is opened, the periods of the controller and gate objects coincide. Assuming that a train having traversed the crossing does not reenter the crossing within the same period, the periods of train objects also coincide with those of the controller and the gate.

We model the behavior of the environmental and system entities using generic reactive classes. For each of the GRC classes *Controller*, *Train*, and *Gate*, we introduce a UML class diagram and a UML statechart

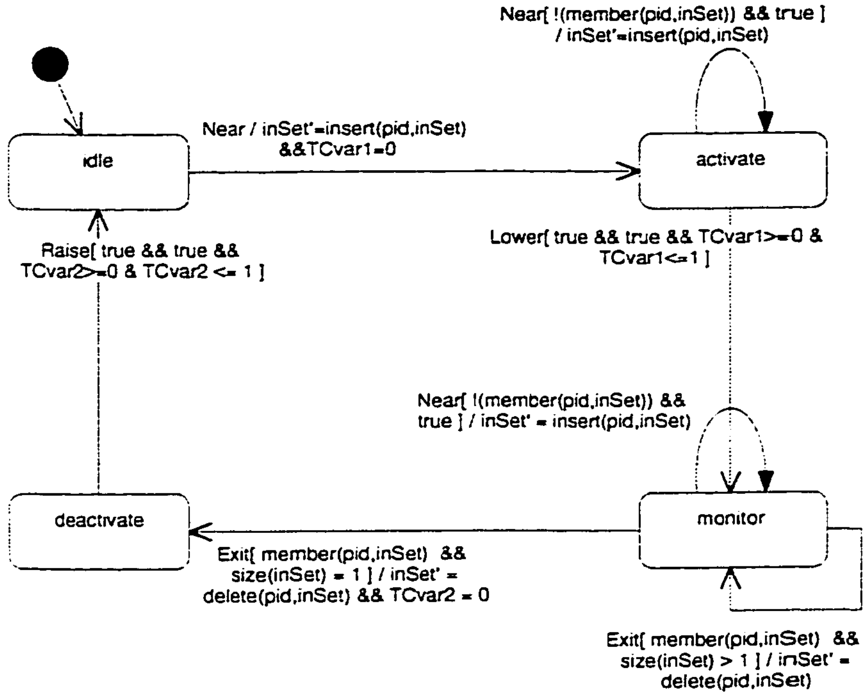


Figure 36: Statechart Diagram for GRC Controller.

diagram, and derive the corresponding formal specifications. Figure 35 shows the GRC classes *Controller*, *Train*, and *Gate*, with their corresponding PortType classes. The binary associations between the PortType classes indicate communication channels between instances of the GRC classes. Figures 36, 37 and 38 show the statechart diagrams depicting the behavior of instances of the GRC classes *Controller*, *Train*, and *Gate*, respectively.

The UML collaboration diagram in Figure 39 shows the configuration of a railroad crossing system with four trains, one controller, and one gate. In this configuration, all the trains interact with the controller. This schematic illustration conforms to a system with trains on distinct routes traversing a crossing. The links between the ports of the reactive objects indicate the message passing mechanism between communicating objects. An analysis of the collaboration diagram, together with the statechart diagrams for the generic reactive classes, provides insight into the overall behavior of the system. The UML sequence diagram in Figure 40 shows a generic scenario for this system. The diagram depicts the timing constraints specified in the statechart diagrams. The logical variables a, b, c, d, e, f, g, and h indicate absolute times for event occurrences. Figures 41, 42 and 43 show the formal specifications generated from the UML model of the generic reactive entities. The LSL trait describing the data model *Set* is available in [GH93]. The formal specifications in Figure 44 describe the configuration of this instance of the railroad crossing system.

6.3.2 Axiomatic Description

In providing an axiomatic description of the design specifications, we define the following sets of events for the respective generic reactive classes.

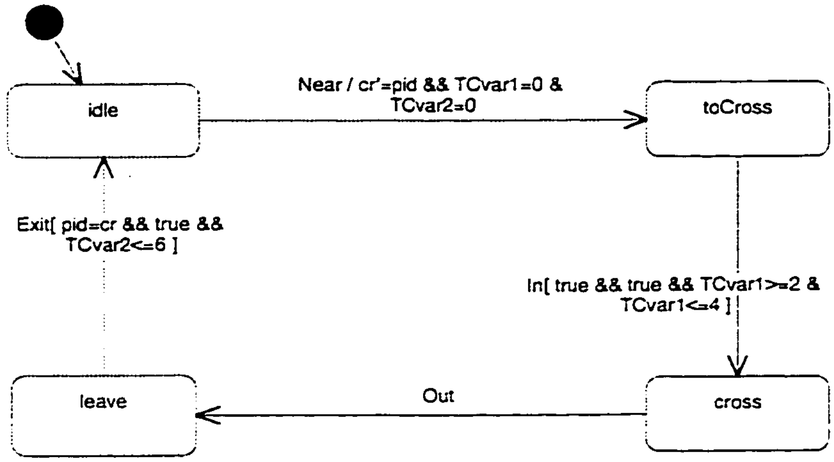


Figure 37: Statechart Diagram for GRC Train.

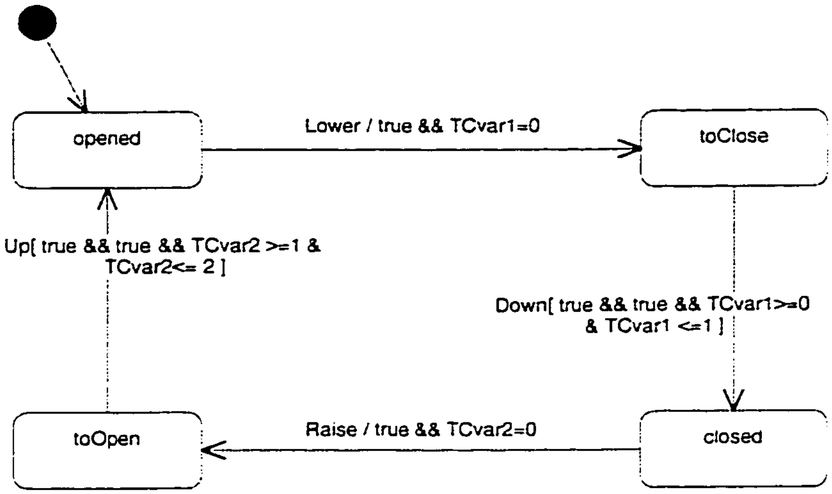


Figure 38: Statechart Diagram for GRC Gate.

```
Train_Event: TYPE
    = {e_Near, e_In, e_Out, e_Exit}
Controller_Event: TYPE
    = {e_Near, e_Exit, e_Lower, e_Raise}
Gate_Event: TYPE
    = {e_Lower, e_Raise, e_Down, e_Up}
```

For each class, we define a higher-order function giving the transition time for an event occurrence, within a period, for an instance of the class. We overload the function *TT* by instantiating the theory *transition_time* containing the function definition. Each instantiation uses the object and event types of the class as actual parameters to the theory. The signature of the functions are as follows.

```
TT: [Train_GRC -> [Period
    -> [Train_Event -> [Occurrence -> Time]]]]
TT: [Controller_GRC -> [Period
```

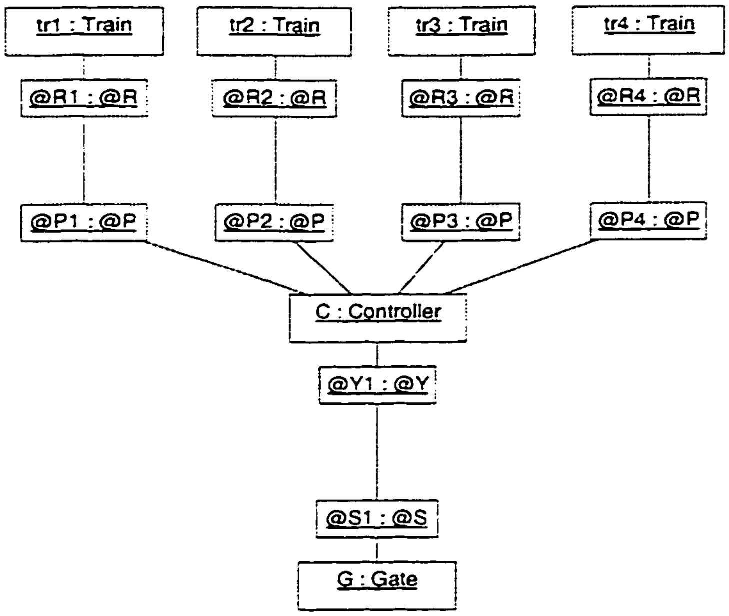


Figure 39: Collaboration Diagram for Railroad Crossing.

```
     -> [Controller_Event -> [Occurrence -> Time]]]]
TT: [Gate_GRC -> [Period
     -> [Gate_Event -> [Occurrence -> Time]]]]
```

We use the variables *i*, *j*, and *tr* as follows: *i* denotes the *i*–th period in the computation of an object, *j* denotes the *j*–th occurrence of an event, and *tr* denotes an instance of the class *Train*. The constants *C* and *G* denote the controller and gate objects, respectively. For instance, *TT(C)(i)(e_Near)(j)* gives the transition time for the *j*–th occurrence of the event *Near* within the *i*–th period in the computation of the controller object *C*.

6.3.3 Transition Axioms

We specify *transition* axioms capturing ordering relations on event occurrences within a period in the computation of an object. We ignore relations on event occurrences across periods since they are not relevant to the safety property of the railroad crossing system. We observe that there is only one occurrence of each of the events *Near*, *In*, *Out*, and *Exit* within a period in the computation of a *Train* object. Similarly, there is only one occurrence of each of the events *Lower*, *Down*, *Raise*, and *Up* within a period in the computation of a *Gate* object. In the computation of a *Controller* object, there is only one occurrence of the events *Lower* and *Raise* within a period; however, there can be multiple occurrences of the events *Near* and *Exit* within the same period.

- The occurrence of the event *Near* precedes the occurrence of the event *In*, within a period *i*, for a train *tr*.

```
TR_AX_1: AXIOM
    TT(tr)(i)(e_Near)(1)
        < TT(tr)(i)(e_In)(1)
```

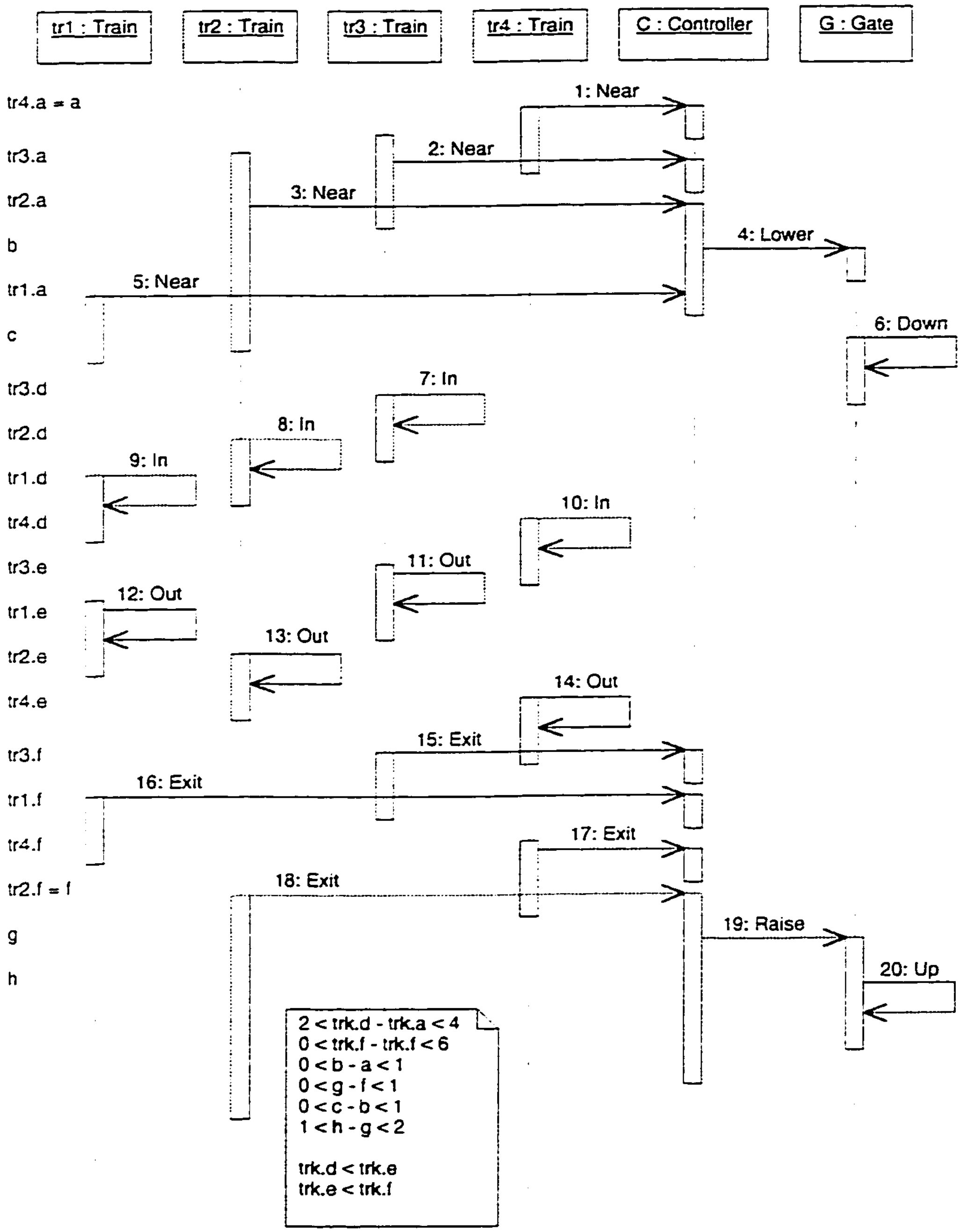


Figure 40: Sequence Diagram for Railroad Crossing.

```
Class Controller [@P, @Y]
    Events: Lower!@Y, Near?@P, Raise!@Y, Exit?@P
    States: *idle, activate, deactivate, monitor
    Attributes: inSet:PSet
    Traits: Set[@P,PSet]
    Attribute-Function:
        activate -> {inSet}; deactivate -> {inSet};
        monitor -> {inSet}; idle -> {};
    Transition-Specifications:
        R1: <activate,monitor>; Lower(true);
            true => true;
        R2: <activate,activate>; Near(NOT(member(pid,inSet)));
            true => inSet' = insert(pid,inSet);
        R3: <deactivate,idle>; Raise(true);
            true => true;
        R4: <monitor,deactivate>; Exit(member(pid,inSet));
            size(inSet) = 1 => inSet' = delete(pid,inSet);
        R5: <monitor,monitor>; Exit(member(pid,inSet));
            size(inSet) > 1 => inSet' = delete(pid,inSet);
        R6: <monitor,monitor>; Near(!(member(pid,inSet)));
            true => inSet' = insert(pid,inSet);
        R7: <idle,activate>; Near(true);
            true => inSet' = insert(pid,inSet);
    Time-Constraints:
        TCvar1: R7, Lower, [0, 1], {};
        TCvar2: R4, Raise, [0, 1], {};
end
```

Figure 41: Formal specification for GRC Controller.

- The occurrence of the event *In* precedes the occurrence of the event *Out*, within a period i, for a train tr.

```
TR_AX_2: AXIOM
    TT(tr)(i)(e_In)(1)
        < TT(tr)(i)(e_Out)(1)
```

- The occurrence of the event *Out* precedes the occurrence of the event *Exit*, within a period i, for a train tr.

```
TR_AX_3: AXIOM
    TT(tr)(i)(e_Out)(1)
        < TT(tr)(i)(e_Exit)(1)
```

- The *first* occurrence of the event *Near* precedes the occurrence of the event *Lower*, within a period i, for the controller C.

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

Figure 42: Formal specification for GRC Train.

```
TR_AX_4: AXIOM
    TT(C)(i)(e_Near)(FirstNear)
        < TT(C)(i)(e_Lower)(1)
```

- The occurrence of the event *Lower* precedes every occurrence *j* of the event *Exit*, within a period *i*, for the controller *C*.

```
TR_AX_5: AXIOM
    TT(C)(i)(e_Lower)(1)
        < TT(C)(i)(e_Exit)(j)
```

- Every occurrence *j* of the event *Exit* precedes the occurrence of the event *Raise*, within a period *i*, for the controller *C*.

```
TR_AX_6: AXIOM
    TT(C)(i)(e_Exit)(j)
        < TT(C)(i)(e_Raise)(1)
```

- The occurrence of the event *Lower* precedes the occurrence of the event *Down*, within a period *i*, for the gate *G*.

```
TR_AX_7: AXIOM
    TT(G)(i)(e_Lower)(1)
        < TT(G)(i)(e_Down)(1)
```

- The occurrence of the event *Down* precedes the occurrence of the event *Raise*, within a period *i*, for the gate *G*.

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

Figure 43: Formal specification for GRC Gate.

```
TR_AX_8: AXIOM
    TT(G)(i)(e_Down)(1)
        < TT(G)(i)(e_Raise)(1)
```

- The occurrence of the event *Raise* precedes the occurrence of the event *Up*, within a period i, for the gate G.

```
TR_AX_9: AXIOM
    TT(G)(i)(e_Raise)(1)
        < TT(G)(i)(e_Up)(1)
```

6.3.4 Time Constraint Axioms

We include a *time constraint* axiom for each constrained reaction to a transition. The occurrence of an event corresponding to a *reaction* happens within a time interval specified relative to the occurrence of the event that activated the reaction. For correct temporal behavior of the railroad crossing controller, the duration between the time of receipt of a message from an approaching train and the time of actual closing of the gate must be smaller than the duration between the sending of the message by the approaching train and the actual entrance of the train into the crossing. In addition, the time at which the gate is reopened must succeed the time at which the controller receives a message from the last train leaving the crossing. The following axioms capture the necessary time constraints.

- The occurrence of the event *In* in reaction to the occurrence of the event *Near*, within a period i, for a train tr, happens within an interval of 2 to 4 time units.

```
SCS TrainGateController
    Includes:
    Instantiate:
        G::Gate[@S:1];
        tr1::Train[@R:1];
        tr2::Train[@R:1];
        tr3::Train[@R:1];
        tr4::Train[@R:1];
        C::Controller[@P:4, @Y:1];
    Configure:
        C.@Y1:@Y <-> G.@S1:@S;
        C.@P1:@P <-> tr1.@R1:@R;
        C.@P2:@P <-> tr2.@R2:@R;
        C.@P3:@P <-> tr3.@R3:@R;
        C.@P4:@P <-> tr4.@R4:@R;
end
```

Figure 44: Formal Specification for Train-Gate-Controller Subsystem.

```
TC_AX_1: AXIOM
    TT(tr)(i)(e_In)(1)
        > TT(tr)(i)(e_Near)(1) + 2
        AND TT(tr)(i)(e_In)(1)
        < TT(tr)(i)(e_Near)(1) + 4
```

- The occurrence of the event *Exit* in reaction to the occurrence of the event *Near*, within a period *i*, for a train *tr*, happens within an interval of 0 to 6 time units.

```
TC_AX_2: AXIOM
    TT(tr)(i)(e_Exit)(1)
        > TT(tr)(i)(e_Near)(1)
        AND TT(tr)(i)(e_Exit)(1)
        < TT(tr)(i)(e_Near)(1) + 6
```

- The occurrence of the event *Lower* in reaction to the *first* occurrence of the event *Near*, within a period *i*, for the controller *C*, happens within an interval of 0 to 1 time unit.

```
TC_AX_3: AXIOM
    TT(C)(i)(e_Lower)(1)
        > TT(C)(i)(e_Near)(FirstNear)
        AND TT(C)(i)(e_Lower)(1)
        < TT(C)(i)(e_Near)(FirstNear) + 1
```

- The occurrence of the event *Raise* in reaction to the *last* occurrence of the event *Exit*, within a period *i*, for the controller *C*, happens within an interval of 0 to 1 time unit.

```
TC_AX_4: AXIOM
    TT(C)(i)(e_Raise)(1)
        > TT(C)(i)(e_Exit)(LastExit)
        AND TT(C)(i)(e_Raise)(1)
        < TT(C)(i)(e_Exit)(LastExit) + 1
```

- The occurrence of the event *Down* in reaction to the occurrence of the event *Lower*, within a period i, for the gate G, happens within an interval of 0 to 1 time unit.

```
TC_AX_5: AXIOM
    TT(G)(i)(e_Down)(1)
        > TT(G)(i)(e_Lower)(1)
        AND TT(G)(i)(e_Down)(1)
        < TT(G)(i)(e_Lower)(1) + 1
```

- The occurrence of the event *Up* in reaction to the occurrence of the event *Raise*, within a period i, for the gate G, happens within an interval of 1 to 2 time units.

```
TC_AX_6: AXIOM
    TT(G)(i)(e_Up)(1)
        > TT(G)(i)(e_Raise)(1) + 1
        AND TT(G)(i)(e_Up)(1)
        < TT(G)(i)(e_Raise)(1) + 2
```

6.3.5 Synchrony Axioms

A synchronized message involves an occurrence of *input* event e in an instance of a reactive class, and an occurrence of *output* event e in an instance of another reactive class. The message causes a transition to occur simultaneously in the two communicating objects. In the railroad crossing system, trains must inform controllers of their intent to traverse a crossing, and of their departure from the crossing. With a communication channel between every train object and the controller object, the occurrence of the output events *Near* and *Exit* in a train object is synchronized with an occurrence of the respective input event in the controller object. A controller synchronizes with the associated gate object, through the events *Lower* and *Raise*, to transmit instructions for closing and opening.

- Within a period i, the occurrence of the event *Near* in every train tr is synchronized with an occurrence j of the event *Near* in the controller C; and, within a period i, every occurrence j of the event *Near* in the controller C is synchronized with the occurrence of the event *Near* in a train tr.

```
SY_AX_1: AXIOM
    (EXISTS j: TT(C)(i)(e_Near)(j)
        = TT(tr)(i)(e_Near)(1))
    AND
    (EXISTS tr: TT(C)(i)(e_Near)(j)
        = TT(tr)(i)(e_Near)(1))
```

- Within a period *i*, the occurrence of the event *Exit* in every train *tr* is synchronized with an occurrence *j* of the event *Exit* in the controller *C*; and, within a period *i*, every occurrence *j* of the event *Exit* in the controller *C* is synchronized with the occurrence of the event *Exit* in a train *tr*.

```
SY_AX_2: AXIOM
    (EXISTS j: TT(C)(i)(e_Exit)(j)
        = TT(tr)(i)(e_Exit)(1))
    AND
    (EXISTS tr: TT(C)(i)(e_Exit)(j)
        = TT(tr)(i)(e_Exit)(1))
```

- Within a period *i*, the occurrence of the event *Lower* in the controller *C* is synchronized with the occurrence of the event *Lower* in the gate *G*.

```
SY_AX_3: AXIOM
    TT(C)(i)(e_Lower)(1)
        = TT(G)(i)(e_Lower)(1)
```

- Within a period *i*, the occurrence of the event *Raise* in the controller *C* is synchronized with the occurrence of the event *Raise* in the gate *G*.

```
SY_AX_4: AXIOM
    TT(C)(i)(e_Raise)(1)
        = TT(G)(i)(e_Raise)(1)
```

6.3.6 Supplementary Axioms

The *supplementary* axioms capture additional requirements for a correct functional and temporal behavior of the system; these requirements are specific to the application. In the case of the generalized railroad crossing, the controller must instruct the gate to close upon receiving a message from the *first* train approaching the crossing, and must instruct the gate to open upon receiving a message from the *last* train leaving the crossing. The attribute *inset* of the GRC class *Controller* is a set to hold the identifiers of the ports through which the controller receives the *Near* message from trains. When a train leaves the crossing, the corresponding port identifier is removed from the set. *inset* allows identifying the trains inside the crossing, and obtaining the count of the number of trains inside the crossing. When the gate is closed, and the controller is monitoring the gate, the controller is deactivated only when *inset* becomes empty.

- Within a period *i*, the occurrence of the event *Near* in the *first* train approaching the crossing precedes the occurrence of the event *Near* in every other train *tr*.

```
FirstTrain_TR_AX: AXIOM
    (tr /= FirstTrain)
        IMPLIES TT(tr)(i)(e_Near)(1)
        > TT(FirstTrain)(i)(e_Near)(1)
```

- Within a period *i*, the occurrence of the event *Exit* in the *last* train leaving the crossing succeeds the occurrence of the event *Near* in every other train *tr*.

```
LastTrain_TR_AX: AXIOM
    (tr /= LastTrain)
        IMPLIES TT(tr)(i)(e_Exit)(1)
        < TT(LastTrain)(i)(e_Exit)(1)
```

- Within a period i, the *first* occurrence of the event *Near* precedes every other occurrence of the event *Near* in the controller C.

```
FirstNear_TR_AX: AXIOM
    (j /= FirstNear)
        IMPLIES TT(C)(i)(e_Near)(j)
        > TT(C)(i)(e_Near)(FirstNear)
```

- Within a period i, the *last* occurrence of the event *Exit* succeeds every other occurrence of the event *Exit* in the controller C.

```
LastExit_TR_AX: AXIOM
    (j /= LastExit)
        IMPLIES TT(C)(i)(e_Exit)(j)
        < TT(C)(i)(e_Exit)(LastExit)
```

- Within a period i, the *first* occurrence of the event *Near* in the controller C synchronizes with the occurrence of the event *Near* in the *first* train approaching the crossing.

```
FirstNear_SY_AX: AXIOM
    TT(C)(i)(e_Near)(FirstNear)
        = TT(FirstTrain)(i)(e_Near)(1)
```

- Within a period i, the *last* occurrence of the event *Exit* in the controller C synchronizes with the occurrence of the event *Exit* in the *last* train leaving the crossing.

```
LastExit_SY_AX: AXIOM
    TT(C)(i)(e_Exit)(LastExit)
        = TT(LastTrain)(i)(e_Exit)(1)
```

6.3.7 Verification of Safety Property

For a safe functional and temporal behavior of the system, the gate must be closed whenever there is a train inside the crossing. Consider the interval $[\alpha_i, \beta_i]$, such that at time α_i there is no train in the crossing and the first train enters the crossing, and at time β_i a train leaves the crossing and there is no other train still in the crossing. The interval $[\alpha_i, \beta_i]$ represents the period corresponding to the i–th closing of the gate. During the interval $[\alpha_i, \beta_i]$, there may be more than one train inside the crossing, and at any time during the interval there is at least one train inside. The safety property corresponds to the gate remaining closed throughout the interval $[\alpha_i, \beta_i]$. The proper closing of the gate involves that within the i–th period,

$$\Sigma_{j=1}^{2} \ d_i^j \ < \ D_i,$$

where D_i is the delay between the time the controller detects the first train approaching the crossing and the time the first train enters the crossing; these two trains need not be the same one. d_i^j represents the delays

1. for the controller to respond to the first stimulus from an approaching train and instruct the gate to close, and

2. for the gate to respond to the instruction to close from the controller.

The gate must close before the *first* train enters the crossing, and must open after the *last* train leaves the crossing. This entails that within the i–th period, the occurrence of the event *Down* in the gate G precedes the occurrence of the event *In* in *every* train tr, and the occurrence of the event *Up* in the gate G succeeds the occurrence of the event *Out* in *every* train tr. The following theorem formalizes the safety property.

```
safety. THEOREM
    TT(G)(i)(e_Down)(1)
        < TT(tr)(i)(e_In)(1)
        AND TT(tr)(i)(e_Out)(1)
        < TT(G)(i)(e_Up)(1)
```

For a safe design, the theorem must be a logical consequence of the axioms derived from the design. Verifying the safety property using PVS theorem prover involves the following proof steps.

Proof steps:

1. introduce the axioms describing the design specifications.

2. skolemize the universally quantified logical variables for the period i and the train tr in the theorem,

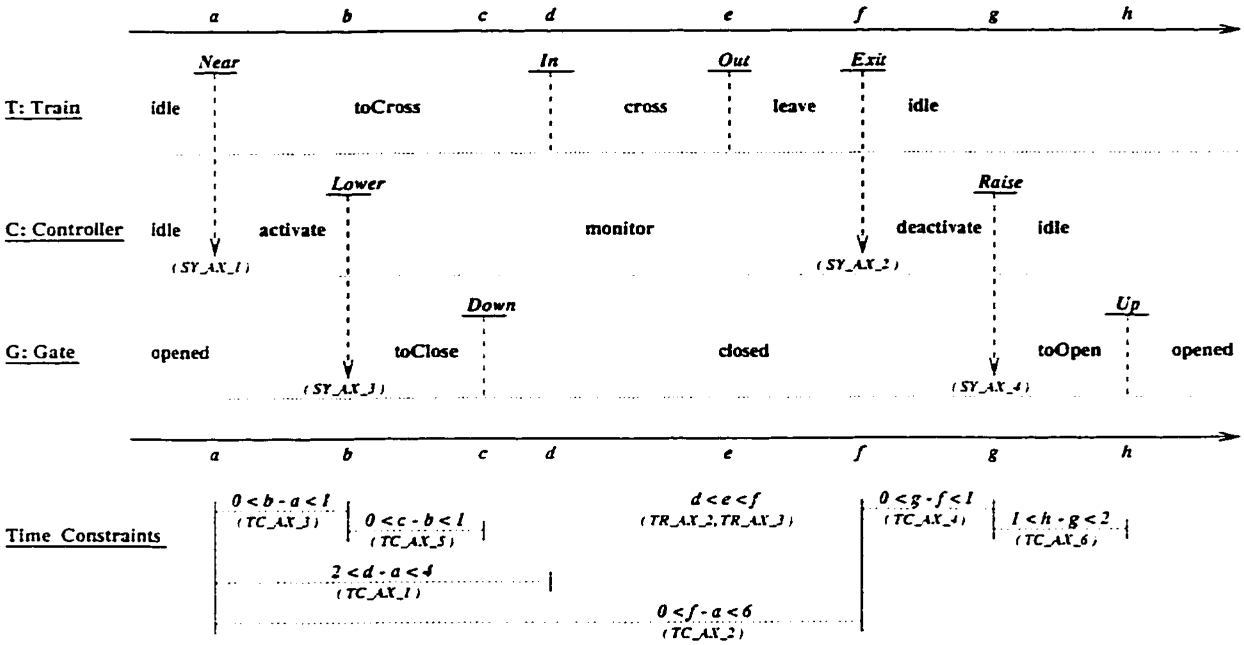
3. instantiate the universally quantified logical variables for the period i and the train tr in the axioms with the respective skolem constants, and

4. apply the *grind* command that instantiates the logical variable j in the axioms with the constants corresponding to the *first* and *last* occurrences of the events *Near* and *Exit*, respectively, in the controller object. The *grind* strategy employs decision procedures for equalities and linear inequalities to assert the formula.

The appendices include the PVS theories for the design specifications of the railroad crossing system, and the PVS commands for proving the safety property.

6.3.8 Proof Correctness

In this section, we establish the correctness of the proof for the safety property of the generalized railroad crossing. We first demonstrate the safety property in the design of a simple railroad crossing system with one train, one controller, and one gate, with a formal proof in PVS. We then show that the design of the generalized railroad crossing system with an arbitrary number of trains satisfies the safety property. The exercise consists in demonstrating that the timing constraints included in the design are consistent with the safety requirement. We use *Time Sequence Charts* to give a view of object interaction in a subsystem.



Legend : a, b, c, d, e, f, g, h : absolute times

Figure 45: Time Sequence Chart for Simple Railroad Crossing.

A *Time Sequence Chart* is a graphical description of a general scenario with transitions and synchronized messages consistent the with time constraints. With absolute time variables on the x-axis, and reactive objects on the y-axis, the graph shows time intervals corresponding to durations for which an object resides in specific states, as well as event occurrences causing state transitions. The chart represents the progression of a flattened subsystem superimposed with the states of the objects, and is a synthesis of the sequence diagram and the statechart diagrams. A time sequence chart facilitates the derivation of axioms involving state predicates and durations, as well as those involving absolute times for event occurrences.

Safety Property of Simple Railroad Crossing System

The time sequence chart in Figure 45 shows a generic scenario for a configuration of the railroad crossing system with one train, one controller, and one gate. The chart uses the logical variables a, b, c, d, e, f, g, and h to indicate absolute times for event occurrences, and conforms to the time constraints in the design specifications. Figure 46 shows the logical clocks for the train, controller, and gate objects. Each object A has two clocks, labeled $A.TCvar1$ and $A.TCvar2$. A clock is initialized to zero when the transition causing the reaction is triggered. The assertions on the value of the clocks determine the interval during which the reaction can occur. The set of axioms in Table 47 correspond to the timing constraints. We use the same naming convention for the axioms, as used in the set of axioms based on the transition time function. The variables a, b, f, and g capture the synchronized occurrence of events in the communicating objects. Similarly, we specify an ordering relation on the occurrence of the events *Out* and *Exit* in the train.

TR_AX_3: e < f

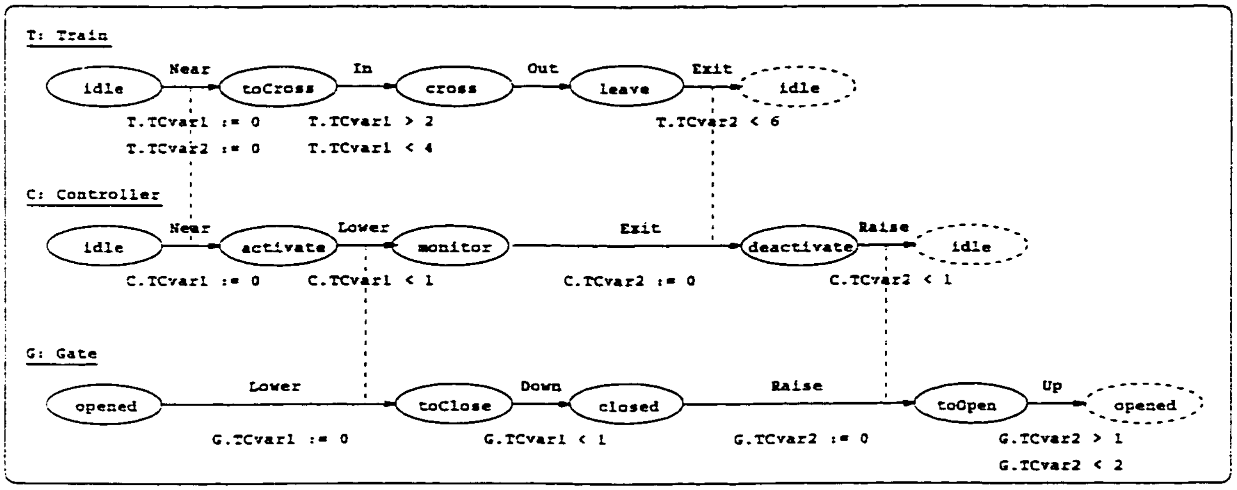


Figure 46: Logical Clocks for Specifying Timing Constraints.

TC_AX_1: d - a > 2 AND d - a < 4
TC_AX_2: f - a > 0 AND f - a < 6
TC_AX_3: b - a > 0 AND b - a < 1
TC_AX_4: g - f > 0 AND g - f < 1
TC_AX_5: c - b > 0 AND c - b < 1
TC_AX_6: h - g > 1 AND h - g < 2

Figure 47: Time-Constraint Axioms for Simple Railroad System.

The proof consists in showing that the gate is closed before the train enters the crossing, that is, $c < d$, and that the gate is opened after the train leaves the crossing, that is, $e < g$. Figure 48 shows the proof steps for the first inequality; the second inequality follows from transition axiom TR_AX_3 and time constraint axiom TC_AX_4. The proof can be mechanically checked as follows. A PVS theory contains axioms capturing the synchronized transitions and time constraints on the behavior of the system. The safety property is expressed as a theorem involving the logical variables. Figure 49 shows the PVS theory of linear inequalities for the design of the simple railroad crossing controller: the PVS proof command *grind* asserts the theorem.

Safety Property of Generalized Railroad Crossing System

For the generalized railroad crossing problem with an arbitrary number of trains, we partition the set of trains twice based on the following criteria: the source and destination states of transitions in the underlying state machine of the controller corresponding to (i) occurrences of the event *Near*, and (ii) occurrences of the event *Exit*. A controller recognizes an approaching train in one of the states *idle*, *activate*, and *monitor*. We partition the set of trains into three disjoint subsets Tr_1, Tr_2, and Tr_3, according to the current state of the controller when it receives the message *Near* from the train. The equivalence class Tr_1 corresponds to the trains from which the controller receives the message in the state *idle*. Tr_1 is a singleton set since the controller transits

1.	From TC_AX_1:	$d-a > 2$
2.	From TC_AX_3:	$b-a < 1$
3.	From TC_AX_5:	$c-b < 1$
4.	From (2) and (3):	$c < b+1 < a+2$
5.	From (1):	$a+2 < d$
6.	From (4) and (5):	$c < d$

Figure 48: Proof Steps for Safety Property.

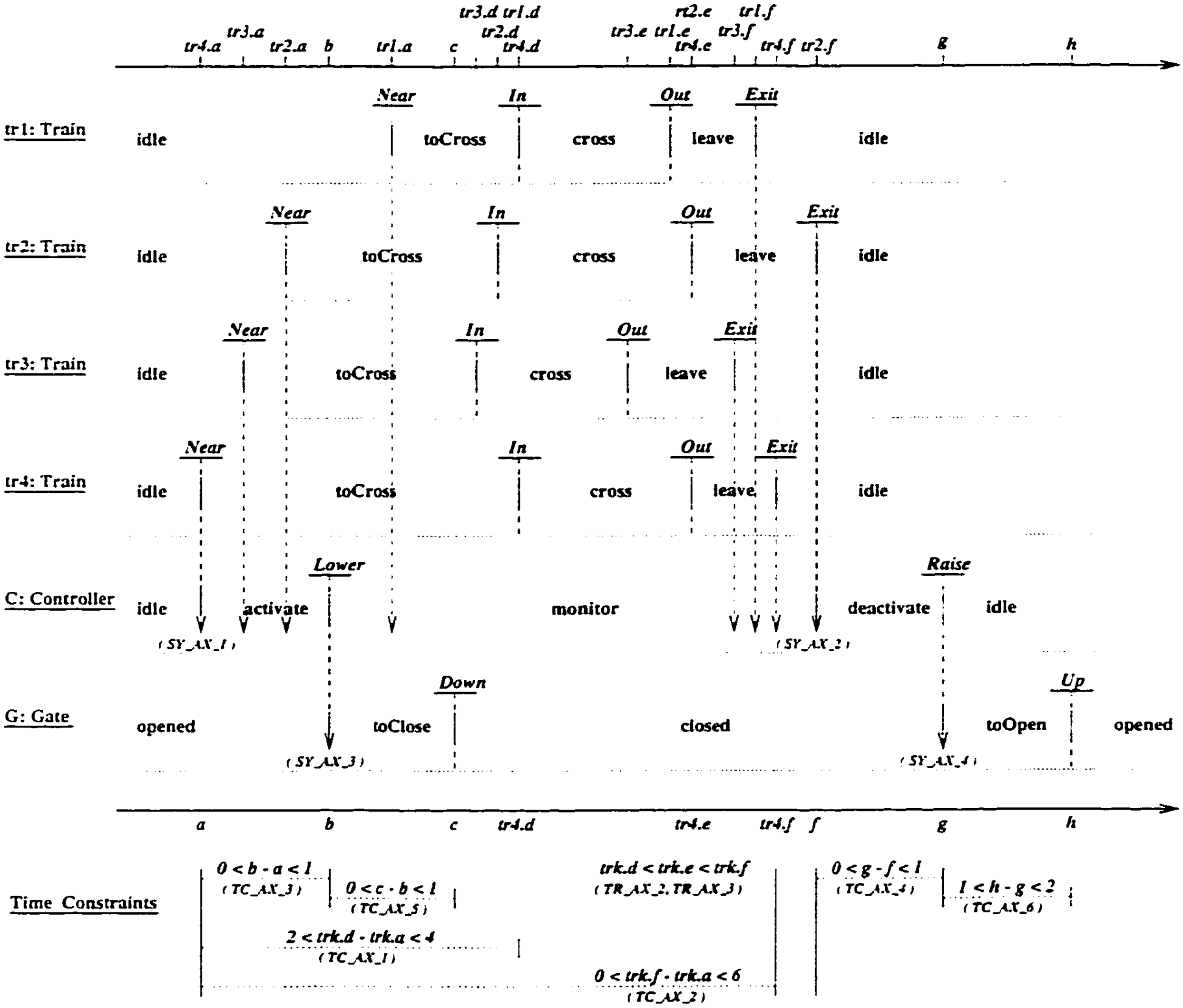
```
tgc: THEORY
BEGIN
    a, b, c, d, e, f, g, h: real
    TC_AX_1: AXIOM d − a > 2 AND d − a < 4
    TC_AX_2: AXIOM f − a > 0 AND f − a < 6
    TC_AX_3: AXIOM b − a > 0 AND b − a < 1
    TC_AX_4: AXIOM g − f > 0 AND g − f < 1
    TC_AX_5: AXIOM c − b > 0 AND c − b < 1
    TC_AX_6: AXIOM h − g > 1 AND h − g < 2
    TR_AX_3: AXIOM e < f
    safety: THEOREM c < d AND e < g
END tgc
```

Figure 49: PVS Specifications for Safety Property.

to the state *activate* on receiving the message. Set Tr_2 corresponds to the trains that send the message *Near* when the controller is in the state *activate*, and set Tr_3 corresponds to the trains that send the message *Near* when the controller is in the state *monitor*.

Similarly, the controller accepts messages from leaving trains while in the state *monitor*. Partitioning the set of all trains according to the time at which the controller receives the message *Exit* from the train, we obtain two equivalence classes Tr_4, and Tr_5. The equivalence class Tr_4 corresponds to the trains that are not the last one to leave the crossing. When a train from Tr_4 sends the message *Near*, the controller is in the state *monitor* and stays in this state, since there are other trains in the crossing. When a train from Tr_5 sends the message *Near*, the controller is in the state *monitor* and transits to the state *deactivate*. Tr_5 is a singleton set since the message is from the last train leaving the crossing, and the controller changes state on receiving this message. The number of trains inside the crossing is obtained from the attribute *inset*, an instance of the trait *Set* holding the port identifiers for messages received from trains entering the crossing. When the controller is monitoring the closed gate, it is deactivated only when *inset* becomes empty.

The time sequence chart in Figure 50 shows a generic scenario for a configuration of the railroad crossing system with four trains, one controller, and one gate. The chart uses the logical variables a, b, c, d, e, f, g,



Legend :

a, b, c, d, e, f, g, h : absolute times

tr1.a, tr2.a, tr3.a, tr4.a, tr1.d, tr2.d, tr3.d, tr4.d : absolute times

tr1.e, tr2.e, tr3.e, tr4.e, tr1.f, tr2.f, tr3.f, tr4.f : absolute times

trk: the variable k can be substituted with 1, 2, 3, or 4 in the formula.

Figure 50: Time Sequence Chart for Generalized Railroad Crossing.

and h to indicate absolute times for event occurrences, and conforms to the time constraints in the sequence diagram in Figure 40. To distinguish between the transition times for the trains, we use $tr_k.a$ to denote the value of variable a for train tr_k. We observe that the scenario depicted in Figure 50 captures the general case by including objects from each of the equivalence classes of trains. For instance,

$$tr_4 \in Tr_1 \ \wedge \ \{tr_2, tr_3\} \subseteq Tr_2 \ \wedge \ tr_1 \in Tr_3,$$

and

$$\{tr_3, tr_1, tr_4\} \subseteq Tr_4 \ \wedge \ tr_2 \in Tr_5.$$

The behavior of a system handling any number of trains, controllers, and gates can be mapped onto this scenario, provided there is a one-to-one relation between gates and controllers. In the inequalities that follow, the subscript i indicates the i–th *period*, that is, the gate is being closed for the i–th time.

We first prove that the gate is closed before a train enters the crossing. The controller goes in state *activate* when the first train approaches the crossing, that is, when train tr_4 sends the message *Near*. When the controller receives the message *Near* from trains tr_2, and tr_3, it remains in state *activate*; when it receives the message *Near* from train tr_1, it stays in state *monitor*.

- From supplementary axiom *FirstNear_SY_AX*, we observe that

  $$a_i = tr_4.a_i \ \wedge \ a_i \leq tr_k.a_i, \ \forall k \in \{1,2,3,4\}, \qquad \cdots\cdots\cdots \ (1)$$

  where a_i denotes the transition time for the controller from state *idle* to state *activate*, and $tr_k.a_i$ represents the transition time when train tr_k goes from state *idle* to state *toCross*, in the i–th period.

- From time constraint axioms *TC_AX_3* and *TC_AX_5*, we infer that

  $$c_i - a_i < 2, \qquad \cdots\cdots\cdots \ (2)$$

  where c_i denotes the transition time for the gate from state *toClose* to state *closed*, in the i–th period.

- From time constraint axiom *TC_AX_1*, we deduce that

  $$tr_k.d_i - tr_k.a_i > 2, \ \forall k \in \{1,2,3,4\}, \qquad \cdots\cdots\cdots \ (3)$$

  where $tr_k.d_i$ represents the transition time when train tr_k goes from state *toCross* to state *cross*, in the i–th period.

- From (1), (2) and (3), we conclude that

  $$c_i < tr_k.d_i, \ \forall k \in \{1,2,3,4\}. \qquad \cdots\cdots\cdots \ (4)$$

We have thus proved that the gate transits to the state *closed* before any of the trains enters the crossing. The formula labeled *(4)* corresponds to the first assertion in the safety property, expressed in the PVS theorem as follows.

```
TT(G)(i)(e_Down)(1)
  < TT(tr)(i)(e_In)(1).
```

Similarly, we show that the gate remains in the state *closed* as long as there is a train in the crossing. The controller goes in the state *deactivate* when the last train leaves the crossing, that is, when train tr_2 sends the message *Exit*. When the controller receives the message *Exit* from the other trains, tr_3, tr_1, and tr_4, it remains in the state *monitor*. In a realistic situation, the order of trains leaving the crossing is independent of the order in which they enter the crossing. The last train to leave the crossing can be any one of tr_1, tr_2, tr_3, and tr_4. However, this aspect has no bearing on the correctness of the proof.

- From supplementary axiom *LastExit_SY_AX*, we observe that

  $$f_i = tr_2.f_i \;\wedge\; tr_k.f_i \leq f_i, \;\; \forall k \in \{1,2,3,4\}, \qquad \cdots\cdots\cdots \; (5)$$

  where f_i denotes the transition time for the controller from state *monitor* to state *deactivate*, and $tr_k.f_i$ represents the transition time when train tr_k goes from state *leave* to state *idle*, in the i–th period.

- From time constraint axioms *TC_AX_4* and *TC_AX_6*, we deduce that

  $$f_i < h_i, \qquad \cdots\cdots\cdots \; (6)$$

  where h_i denotes the transition time for the gate from state *toOpen* to state *opened*, in the i–th period.

- From transition axiom *TR_AX_3*, we deduce that

  $$tr_k.e_i \leq tr_k.f_i, \;\; \forall k \in \{1,2,3,4\}, \qquad \cdots\cdots\cdots \; (7)$$

  where $tr_k.e_i$ represents the transition time when train tr_k goes from state *cross* to state *leave*, in the i–th period.

- From (5), (6) and (7), we conclude that

  $$tr_k.e_i < h_i, \;\; \forall k \in \{1,2,3,4\}. \qquad \cdots\cdots\cdots \; (8)$$

We have thus proved that the gate enters the state *opened* only after all trains have left the crossing. The formula labeled *(8)* corresponds to the second assertion in the safety property, expressed in the PVS theorem as follows.

```
TT(tr)(i)(e_Out)(1)
  < TT(G)(i)(e_Up)(1)
```

The cases illustrated by trains tr_1, tr_2, tr_3 and tr_4 capture all possible scenarios both for trains entering the crossing, according to the equivalence classes Tr_1, Tr_2 and Tr_3, and for trains leaving the crossing, according to the equivalence classes Tr_4 and Tr_5. This completes the proof of safety property in the design of the generalized railroad crossing system.

6.4 Logical Foundation

In this section, we discuss a logical foundation based on state machine semantics, and informally motivate how we can obtain the axiomatic description of design specifications by applying the duality between state transitions and event occurrences. A behavioral semantics for the object-oriented formalism serves as a foundation for the proposed verification methodology. Achuthan [Ach95] proposed an axiomatization for the abstract reactive model. Since UML metamodel and well-formedness rules are defined in OCL (Object Constraint Language) [OMG99], our endeavor is to obtain a consistent semantic framework within OCL. We incorporate temporal predicates in the language, and provide logical semantics for the abstract real-time reactive model. OCL is a typed functional language supporting first-order formulas for specifying relations and constraints on objects. We obtain the set of axioms for a reactive object by substituting status data for the formal arguments of predicates in the axioms.

In defining the semantics of a subsystem, we view a collaboration diagram as a configuration of instances of generic reactive classes. We introduce a new stereotype, *PortLink*, on binary associations between port objects, to define links between communicating reactive objects. The abstract class, *Subsystem*, is a generalization of the subsystem configuration specifications; it represents a composition of *instances* and *portlinks*. The attribute *instances* defines a set of objects of the generic reactive classes, and the attribute *portlinks* defines a set of binary associations between ports of the reactive objects. A tuple of the form

$$<< instance_1, port_1 >, < instance_2, port_2 >>$$

defines a port-link for a communication channel between port $port_1$ of object $instance_1$ and port $port_2$ of object $instance_2$. In this model, the only form of communication between reactive objects is through synchronous message passing.

6.4.1 Behavioral Semantics

The following subset of axioms defines the behavior of instances of the generic reactive classes: *Transition* axiom, *Constrained-event* axiom, and *Synchrony* axiom. The *Transition* axiom describes the effect of an occurrence of an event. The *Constrained-event* axiom describes the confinement of a reaction to within a time interval subsequent to an event occurrence. The *Synchrony* axiom describes the semantics for synchronous message passing between linked objects. Section 5.7 includes an OCL specification of the axioms.

- *Transition axiom:*

  The transition axiom is defined for each transition specification of a reactive object. The occurrence of an event results in a state transition to the target state, and the satisfaction of the post-condition in the target state. If the target state is a complex state, then the initial state from its set of substates replaces it recursively until the destination is a simple state.

- *Constrained-event axiom:*

  A set of time constraint axioms defines the behavior of a reactive object. The axioms apply for each time-constraint of a generic reactive class. A trigger event is necessary for the occurrence of a reaction in the form of a constrained event.

- *Synchrony axiom:*

  The synchrony axiom applies for each port-link in a *subsystem configuration specification*.

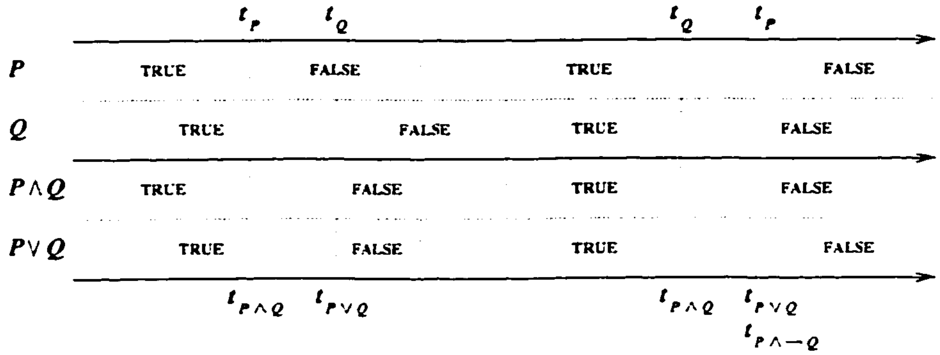


Figure 51: Absolute Times at which a Predicate Becomes *false*.

6.4.2 The *since* Operator

Shankar [Sha93] introduces the *since* operator for measuring durations in the context of reasoning about real-time behavior. At time t, the value of *since* for a predicate P, written *since*(P) or $|P|$, is the time that has elapsed since the predicate P 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 false to true. The *since* operator 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.

We formulate a transformation for formulas involving the *since* operator into linear inequalities over absolute time variables. We use the variable t_P to denote the absolute time at which predicate P was last *true*. Figure 51 illustrates the relation between t_P, t_Q, $t_{P\wedge Q}$, $t_{P\vee Q}$, and $t_{P\wedge\neg Q}$. The variables P and Q range over state predicates. Shankar [Sha92] introduces lemmas capturing invariants on the behavior of the *since* operator. Table 7 shows the lemmas, each with its equivalent in terms of absolute times at which the predicates become *false*.

A predicate formula of the form

$$P \supset |Q| < x$$

where P and Q are state predicates, is transformed into the linear inequality

$$t < t_Q + x$$

for all values of t, such that 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

$$P \supset |Q| < |R| + x$$

we obtain 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*. Using the relation between the *since* operator and absolute times, formulas involving other boolean operators yield similar linear inequalities.

Using *since* operator	Using absolute times
$\{\lvert(\lvert P\rvert \le x)\rvert \le y \supset \lvert P\rvert \le x+y\}$	$t_{\lvert(\lvert P\rvert\le x)\rvert\le y} = t_{\lvert P\rvert\le x} + y$ $= t_{\lvert P\rvert} + x + y$
$\{\lvert P \vee Q\rvert \le \min(\lvert P\rvert, \lvert Q\rvert)\}$	$t_{P\vee Q} = \max(t_P, t_Q)$
$\{\lvert P \vee Q\rvert = \lvert P\rvert \vee \lvert P \vee Q\rvert = \lvert Q\rvert\}$	$t_{P\vee Q} = t_P \;\vee\; t_{P\vee Q} = t_Q$
$\{\lvert P \wedge Q\rvert = \lvert P\rvert \vee \lvert P \wedge \neg Q\rvert = \lvert P\rvert\}$	$t_{P\wedge Q} = t_P \;\vee\; t_{P\wedge\neg Q} = t_P$
$\{\max(\lvert P\rvert, \lvert Q\rvert) \le \lvert P \wedge Q\rvert\}$	$t_{P\wedge Q} = \min(t_P, t_Q)$
$\{\lvert P\rvert \le \lvert P \wedge Q\rvert\}$	$t_P \ge t_{P\wedge Q}$
$\{\lvert Q\rvert \le \lvert P \wedge Q\rvert\}$	$t_Q \ge t_{P\wedge Q}$

Table 7: Properties of the *since* Operator.

6.4.3 Deriving Axioms from Behavioral Semantics

Applying the behavioral semantics to derive axioms from the UML model of a system is a non-trivial exercise. However, an axiomatic description can be obtained by instantiating the axioms defined in the behavioral semantics with data from the formal specifications of the design. We derive axioms based on

- transition specifications, from the *Transition* axiom (*TR*),
- time constraints, from the *Constrained-event* axiom (*CE*) and the *Transition* axiom (*TR*), and
- synchronized messages, from the *Synchrony* axiom (*SY*), and the *Transition* axiom (*TR*) of the behavioral semantics.

The resulting axioms involve durations on state predicates; time intervals are specified using the *since* operator. We note that this set of axioms is *incomplete* since the supplementary axioms described earlier cannot be derived from the behavioral semantics. While transition, time constraint, and synchrony axioms relate to the domain of real-time reactive systems, the supplementary axioms capture additional system-specific requirements for correct functional and temporal behavior. However, the supplementary axioms can be expressed in terms of state predicates and durations. Similarly, we can use state predicates and the *since* operator to specify the property to be verified as an invariance assertion on the state of the system. Based on the relation between the *since* operator and absolute times, it is straightforward to translate the axioms and theorem involving the *since* operator into formulas involving linear inequalities over absolute times to obtain a PVS theory of linear inequalities.

The state of a reactive system, viewed as the *statuses* of objects in the system, includes the current state, assignment vector, and reaction vector of each object. The assignment vector gives the value of each attribute of the object in the current state. The occurrence of an event can lead to reactions in the form of internal or output events occurring within specified time intervals. The reaction vector represents a list of outstanding reactions to transitions. For each generic reactive class, we define a higher-order function whose domain is

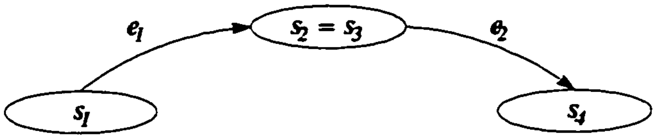


Figure 52: Ordering Relation on Transitions.

the set of instances of the class, and that returns a function taking a *system state* and returning the state of the specified object in the given system state. For instance, $train(tr)(s)$ returns the state of the train object tr in the system state s.

Transition Axioms

A *transition* axiom corresponds to an ordering relation on the occurrence of two valid transitions in the system, that is, an assertion on two system states. The transition axiom *TR* specifies the source and destination states for a transition. Instantiating transition axiom *TR* twice, and assuming that the postconditions hold for the transitions, we obtain the following axioms.

$$\begin{aligned}
&A.HoldAt(s_1,t_1) \text{ and } A.Occur(e_1,p_i,t_1)\\
&\quad \text{and } t_1 < t_2 \text{ implies } A.HoldAt(s_2,t_2).\\
&A.HoldAt(s_3,t_3) \text{ and } A.Occur(e_2,p_i,t_3)\\
&\quad \text{and } t_3 < t_4 \text{ implies } A.HoldAt(s_4,t_4).
\end{aligned}$$

If $s_2 = s_3$, we deduce that when object A is in state s_4, the time since object A left state s_1 is greater than the time since object A left state s_2. The following axiom involving the *since* operator captures this property.

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

The assertion is valid for every state subsequent to state s_4 within the same period in the computation of object A. Figure 52 illustrates the ordering relation on transitions. We note the duality with the event occurrences to assert that an occurrence of event e_1 precedes an occurrence of event e_2, and conclude that this axiom based on durations can be translated into one using the transition time function *TT* as follows.

$$TT(A)(i)(e_1)(j) \; < \; TT(A)(i)(e_2)(j).$$

Time Constraint Axioms

In deriving *time constraint* axioms, we consider the time interval during which a reaction to a transition is to occur. An upper time bound u on the firing of a transition stands for a constraint that if the object is in the source state of the transition, then the time elapsed since the reaction was enabled is less than u. A lower time bound l on the firing of a transition stands for a constraint that if the object is in the destination state of the transition, then the time elapsed since the reaction was enabled is greater than l. Other axioms can be included to capture similar properties in states prior to entering the source state for the upper time bound, and in states subsequent to leaving the destination state for the lower time bound. Applying the constrained-event axiom *CE* on a reactive object A, we deduce that an occurrence of event e_1 triggers event e_4 to occur in reaction.

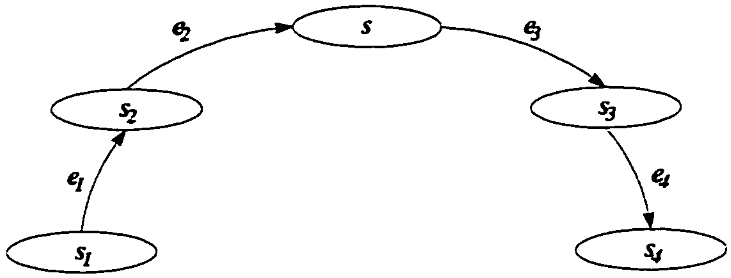


Figure 53: Time Constraint on Reaction.

$A.Occur(e_4, p_i, t_c)$ implies
$A.Occur(e_1, p_j, t_a)$ and $t_a + l \leq t_c$ and $t_c \leq t_a + u$.

Since every event occurrence is associated with a transition, and the transition axiom *TR* specifies the source and destination states for the transition, we derive the following axioms.

$A.HoldAt(s_1, t_1)$ and $A.Occur(e_1, p_i, t_1)$
and $t_1 < t_2$ implies $A.HoldAt(s_2, t_2)$.
$A.HoldAt(s_3, t_3)$ and $A.Occur(e_4, p_i, t_3)$
and $t_3 < t_4$ implies $A.HoldAt(s_4, t_4)$.

If $s_2 = s_3$, we deduce that when object A is in state s_4, the duration between the time object A left state s_1 and the time object A left state s_2 lies in the interval $[l, u]$. The assertion holds in every state subsequent to state s_4 within the same period in the computation of object A. The following axioms involving the *since* operator capture these properties.

$$A = s_4 \supset since(A = s_1) > since(A = s_3) + l \ \wedge$$
$$A = s_4 \supset since(A = s_1) < since(A = s_3) + u.$$

If $s_2 \neq s_3$, in addition to the above properties, the following assertion holds in every intermediate state s between states s_2 and s_3 in the computation of object A.

$$A = s \supset since(A = s_1) < u.$$

Figure 53 illustrates the time constraint on reaction. We observe that this property can be stated in terms of occurrence times for events e_1 and e_4, using the transition time function *TT* as follows.

$$TT(A)(i)(e_4)(j) \ > \ TT(A)(i)(e_1)(j) \ + \ l \ \wedge$$
$$TT(A)(i)(e_4)(j) \ < \ TT(A)(i)(e_1)(j) \ + \ u.$$

Synchrony Axioms

We derive *synchrony* axioms from the transition specifications for occurrences of *external* events. A synchrony axiom describes a simultaneous change of state for the two communicating objects. For each external event, and for each pair of communicating objects, we instantiate the transition axiom *SY*, and the synchrony

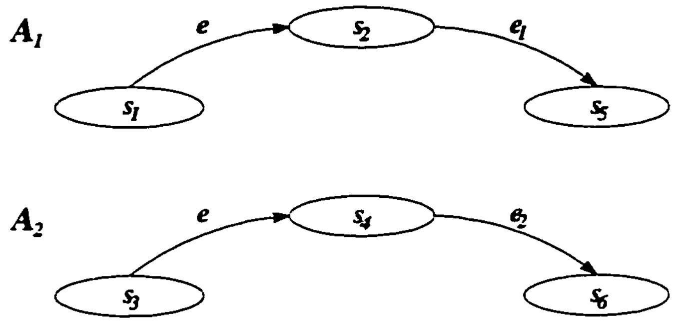


Figure 54: Synchronized Transitions.

axiom *TR*. The duration since the occurrence of the synchronized event will be the same for both objects in the respective destination states. In addition, this assertion holds in all subsequent states prior to completion of the period for one of the communicating objects. Instantiating the synchroy axiom *SY*, we obtain the following assertions.

$A_1.Occur(e, p_1, t)$
 implies $A_2.Occur(e, p_2, t)$.
$A_2.Occur(e, p_2, t)$
 implies $A_1.Occur(e, p_1, t)$.

Since event occurrences are associated with transitions, and the transition axiom *TR* specifies the source and destination states for the transition, we derive the following axioms.

$A_1.HoldAt(s_1, t_1)$ and $A_1.Occur(e, p_i, t_1)$
 and $t_1 < t_2$ implies $A_1.HoldAt(s_2, t_2)$.
$A_2.HoldAt(s_3, t_3)$ and $A_2.Occur(e, p_i, t_3)$
 and $t_3 < t_4$ implies $A_2.HoldAt(s_4, t_4)$.

If $t_1 = t_3$, we deduce that when object A_1 is in state s_2 and object A_2 is in state s_4, the time since object A_1 left state s_1 is equal to the time since object A_2 left state s_3. The following axiom involving the *since* operator captures this property.

$$A_1 = s_2 \wedge A_2 = s_4 \supset since(A_1 = s_1) = since(A_2 = s_3).$$

The assertion holds in every pair of states subsequent to states s_2 and s_4, within the same period in the computations of objects A_1 and A_2, respectively. Figure 54 illustrates the synchronization of transitions. The property can be stated in terms of an equality on occurrence times for event e in objects A_1 and A_2, using the transition time function *TT* as follows.

$$TT(A_1)(i)(e)(j) = TT(A_2)(i)(e)(j).$$

6.5 Axiom Derivation for the Case Study

The goal of this section is to show that axioms based on durations derived from the behavioral semantics correspond to axioms derived using the transition time function *TT*. We apply the semantics to the generalized railroad crossing system to extract axioms based on state predicates and durations. Since the analysis is based on state predicates, the assignment vector and the reaction vector of the reactive objects become irrelevant and are omitted in the axioms. The configuration of the system includes a port-link between each train and the controller, and one port-link between the controller and the gate. We use variable s_i to denote a system state: an expression of the form $train(tr)(s_i)$ gives the projection of the state s_i on the *train* object tr. We use the following naming convention for the axioms based on durations: for all n, axioms *AX_m.n* together correspond to the axiom *AX_m* expressed in terms of the transition time function.

6.5.1 Transition Axioms

Substituting states and event occurrences in the transition axiom *TR* of the logical semantics, we apply the algorithms described in the previous section to derive axioms describing the transition specifications using the *since* operator. The transition axioms described earlier using the function *TT* can be obtained from the axioms in this section.

- Train tr enters the crossing - internal event *In* - after sending the message *Near* to the controller.

  TR_AX_1.1: $\text{train(tr)(s0)} = \text{cross} \supset$
  $\quad \text{since(train(tr)(s1)} = \text{idle)} > \text{since(train(tr)(s2)} = \text{toCross)}$
  TR_AX_1.2: $\text{train(tr)(s0)} = \text{leave} \supset$
  $\quad \text{since(train(tr)(s1)} = \text{idle)} > \text{since(train(tr)(s2)} = \text{toCross)}$

- Train tr leaves the crossing - internal event *Out* - after the internal event *In*.

  TR_AX_2.1: $\text{train(tr)(s0)} = \text{leave} \supset$
  $\quad \text{since(train(tr)(s1)} = \text{toCross)} > \text{since(train(tr)(s2)} = \text{cross)}$
  TR_AX_2.2: $\text{train(tr)(s0)} = \text{idle} \supset$
  $\quad \text{since(train(tr)(s1)} = \text{toCross)} > \text{since(train(tr)(s2)} = \text{cross)}$

- Train tr sends the message *Exit* to the controller after leaving the crossing - internal event *Out*.

  TR_AX_3.1: $\text{train(tr)(s0)} = \text{idle} \supset$
  $\quad \text{since(train(tr)(s1)} = \text{cross)} > \text{since(train(tr)(s2)} = \text{leave)}$

- Controller c sends the message *Lower* to the gate after receiving the message *Near* from the first approaching train.

  TR_AX_4.1: $\text{controller(C)(s0)} = \text{monitor} \supset$
  $\quad \text{since(controller(C)(s1)} = \text{idle)} >$

since(controller(C)(s2) = activate)

TR_AX_4.2: controller(C)(s0) = deactivate $\supset$
since(controller(C)(s1) = idle) $>$
since(controller(C)(s2) = activate)

- Controller *c* receives the message *Exit* from the last train leaving the crossing after sending the message *Lower* to the gate.

TR_AX_5.1: controller(C)(s0) = deactivate $\supset$
since(controller(C)(s1) = activate) $>$
since(controller(C)(s2) = monitor)

TR_AX_5.2: controller(C)(s0) = idle $\supset$
since(controller(C)(s1) = activate) $>$
since(controller(C)(s2) = monitor)

- Controller *c* sends the message *Raise* to the gate after receiving the message *Exit* from the last train leaving the crossing.

TR_AX_6.1: controller(C)(s0) = idle $\supset$
since(controller(C)(s1) = monitor) $>$
since(controller(C)(s2) = deactivate)

- Gate *g* closes - internal event *Down* - after receiving the message *Lower* from the controller.

TR_AX_7.1: gate(G)(s0) = closed $\supset$
since(gate(G)(s1) = opened) $>$ since(gate(G)(s2) = toClose)

TR_AX_7.2: gate(G)(s0) = toOpen $\supset$
since(gate(G)(s1) = opened) $>$ since(gate(G)(s2) = toClose)

- Gate *g* receives the message *Raise* from the controller after closing - internal event *Down*.

TR_AX_8.1: gate(G)(s0) = toOpen $\supset$
since(gate(G)(s1) = toClose) $>$ since(gate(G)(s2) = closed)

TR_AX_8.2: gate(G)(s0) = opened $\supset$
since(gate(G)(s1) = toClose) $>$ since(gate(G)(s2) = closed)

- Gate *g* opens - internal event *Up* - after receiving the message *Raise* from the controller.

TR_AX_9.1: gate(G)(s0) = opened $\supset$
since(gate(G)(s1) = closed) $>$ since(gate(G)(s2) = toOpen)

6.5.2 Time Constraint Axioms

We instantiate the constrained-event axiom *CE* and the transition axiom *TR* with the time-constrained transitions in the design specifications, and apply the algorithms described in the previous section to express the properties using the *since* operator. The time constraint axioms described earlier using the function *TT* can be obtained from the axioms in this section.

- Train *tr* enters the crossing - internal event *In* - within a window of 2 to 4 time units after sending the message *Near* to the controller.

  TC_AX_1.1: train(tr)(s0) = cross ⊃
  since(train(tr)(s1) = idle) > 2
  TC_AX_1.2: train(tr)(s0) = cross ⊃
  since(train(tr)(s1) = idle) > since(train(tr)(s2) = toCross) + 2
  TC_AX_1.3: train(tr)(s0) = leave ⊃
  since(train(tr)(s1) = idle) > since(train(tr)(s2) = toCross) + 2
  TC_AX_1.4: train(tr)(s0) = toCross ⊃
  since(train(tr)(s1) = idle) < 4
  TC_AX_1.5: train(tr)(s0) = cross ⊃
  since(train(tr)(s1) = idle) < since(train(tr)(s2) = toCross) + 4
  TC_AX_1.6: train(tr)(s0) = leave ⊃
  since(train(tr)(s1) = idle) < since(train(tr)(s2) = toCross) + 4

- Train *tr* sends the message *Exit* to the controller within 6 time units after sending the message *Near* to the controller.

  TC_AX_2.1: train(tr)(s0) = leave ⊃
  since(train(tr)(s1) = idle) < 6
  TC_AX_2.2: train(tr)(s0) = cross ⊃
  since(train(tr)(s1) = idle) < 6
  TC_AX_2.3: train(tr)(s0) = toCross ⊃
  since(train(tr)(s1) = idle) < 6

- Controller *c* sends the message *Lower* to the gate within 1 time unit after receiving the message *Near* from the first approaching train.

  TC_AX_3.1: controller(C)(s0) = activate ⊃
  since(controller(C)(s1) = idle) < 1
  TC_AX_3.2: controller(C)(s0) = monitor ⊃
  since(controller(C)(s1) = idle) <
  since(controller(C)(s2) = activate) + 1
  TC_AX_3.3: controller(C)(s0) = deactivate ⊃
  since(controller(C)(s1) = idle) <
  since(controller(C)(s2) = activate) + 1

- Controller *c* sends the message *Raise* to the gate within 1 time unit after receiving the message *Exit* from the last train leaving the crossing.

TC_AX_4.1: controller(C)(s0) = deactivate ⊃
  since(controller(C)(s1) = monitor) < 1
TC_AX_4.2: controller(C)(s0) = idle ⊃
  since(controller(C)(s1) = monitor) <
    since(controller(C)(s2) = deactivate) + 1

- Gate *g* closes - internal event *Down* - within 1 time unit after receiving the message *Lower* from the controller.

TC_AX_5.1: gate(G)(s0) = toClose ⊃
  since(gate(G)(s1) = opened) < 1
TC_AX_5.2: gate(G)(s0) = closed ⊃
  since(gate(G)(s1) = opened) < since(gate(G)(s2) = toClose) + 1
TC_AX_5.3: gate(G)(s0) = toOpen ⊃
  since(gate(G)(s1) = opened) < since(gate(G)(s2) = toClose) + 1

- Gate *g* opens - internal event *Up* - within a window of 1 to 2 time units after receiving the message *Raise* from the controller.

TC_AX_6.1: gate(G)(s0) = opened ⊃
  since(gate(G)(s1) = closed) > 1
TC_AX_6.2: gate(G)(s0) = opened ⊃
  since(gate(G)(s1) = closed) > since(gate(G)(s2) = toOpen) + 1
TC_AX_6.3: gate(G)(s0) = toOpen ⊃
  since(gate(G)(s1) = closed) < 2
TC_AX_6.4: gate(G)(s0) = opened ⊃
  since(gate(G)(s1) = closed) < since(gate(G)(s2) = toOpen) + 2

6.5.3 Synchrony Axioms

Object communication is subject to synchronous external events. Hence, communicating objects enter states concurrently in synchronized transitions. We instantiate the transition axiom *TR* and the synchrony axiom *SY* with data from the design specifications, and apply the algorithms described in the previous section to rewrite the assertions using the *since* operator. The synchrony axioms described earlier using the function *TT* can be obtained from the axioms in this section.

- Train *tr* synchronizes with controller *c* for the message *Near*. The *first* approaching train, tr_4, causes a change of state in the controller.

SY_AX_1.1:

$\text{train(tr4)(s0)} = \text{toCross} \land \text{controller(C)(s0)} = \text{activate} \supset$
$\quad \text{since(train(tr4)(s1)} = \text{idle)} = \text{since(controller(C)(s2)} = \text{idle)}$

SY_AX_1.2:

$\text{train(tr4)(s0)} = \text{toCross} \land \text{controller(C)(s0)} = \text{monitor} \supset$
$\quad \text{since(train(tr4)(s1)} = \text{idle)} = \text{since(controller(C)(s2)} = \text{idle)}$

SY_AX_1.3:

$\text{train(tr4)(s0)} = \text{cross} \land \text{controller(C)(s0)} = \text{monitor} \supset$
$\quad \text{since(train(tr4)(s1)} = \text{idle)} = \text{since(controller(C)(s2)} = \text{idle)}$

SY_AX_1.4:

$\text{train(tr4)(s0)} = \text{leave} \land \text{controller(C)(s0)} = \text{monitor} \supset$
$\quad \text{since(train(tr4)(s1)} = \text{idle)} = \text{since(controller(C)(s2)} = \text{idle)}$

- Train tr synchronizes with controller c for the message *Exit*. The *last* train leaving, tr_2, causes a change of state in the controller.

SY_AX_2.1:

$\text{train(tr2)(s0)} = \text{idle} \land \text{controller(C)(s0)} = \text{deactivate} \supset$
$\quad \text{since(train(tr2)(s1)} = \text{leave)} = \text{since(controller(C)(s2)} = \text{monitor)}$

SY_AX_2.2:

$\text{train(tr2)(s0)} = \text{idle} \land \text{controller(C)(s0)} = \text{idle} \supset$
$\quad \text{since(train(tr2)(s1)} = \text{leave)} = \text{since(controller(C)(s2)} = \text{monitor)}$

- Controller c synchronizes with gate g for the message *Lower*.

SY_AX_3.1:

$\text{controller(C)(s0)} = \text{monitor} \land \text{gate(G)(s0)} = \text{toClose} \supset$
$\quad \text{since(controller(C)(s1)} = \text{activate)} = \text{since(gate(G)(s2)} = \text{opened)}$

SY_AX_3.2:

$\text{controller(C)(s0)} = \text{monitor} \land \text{gate(G)(s0)} = \text{closed} \supset$
$\quad \text{since(controller(C)(s1)} = \text{activate)} = \text{since(gate(G)(s2)} = \text{opened)}$

SY_AX_3.3:

$\text{controller(C)(s0)} = \text{deactivate} \land \text{gate(G)(s0)} = \text{closed} \supset$
$\quad \text{since(controller(C)(s1)} = \text{activate)} = \text{since(gate(G)(s2)} = \text{opened)}$

- Controller c synchronizes with gate g for the message *Raise*.

SY_AX_4.1:

$\text{controller(C)(s0)} = \text{idle} \land \text{gate(G)(s0)} = \text{toOpen} \supset$
$\quad \text{since(controller(C)(s1)} = \text{deactivate)} = \text{since(gate(G)(s2)} = \text{closed)}$

SY_AX_4.2:

$\text{controller(C)(s0)} = \text{idle} \land \text{gate(G)(s0)} = \text{opened} \supset$
$\quad \text{since(controller(C)(s1)} = \text{deactivate)} = \text{since(gate(G)(s2)} = \text{closed)}$

Chapter 7

Conclusions and Future Work

We have made three significant contributions to the development of real-time reactive systems. These pertain to RTUML notation for visual modeling, formal semantics for RTUML, and mechanized verification of systems described in RTUML. The visual modeling technique is based on UML notation, with minimal extensions and well-formedness rules for syntactic correctness. The formal semantics serves as a foundation for design, specification, validation and verification methods. The verification methodology, developed within the PVS verification environment, applies to time-dependent properties, including safety and liveness properties. Future goals include extending the methodology to support parameterized events and inheritance.

7.1 Conclusions

The thesis includes three significant contributions for the development of real-time reactive systems:

- RTUML – Real-Time UML.
- Formal semantics of RTUML,
- Mechanized verification methodology.

7.1.1 RTUML – Real-Time UML

We have presented a framework for using UML in the development process of complex real-time reactive systems. The motivation for our approach relies on the distinction of a user-level design notation from the more formal notation necessary for rigorous analysis of requirement specifications. This discrimination has several merits.

1. Existing UML tools can be used for modeling.
2. The well-formedness of design models can be checked using a tool built in conformance with the semantics described in OCL.
3. The correctness of a design with respect to its satisfaction of desired system properties can be formally verified.

We have developed a UML graphic model capturing the concepts included in an abstract model of a real-time reactive system, using a layered approach with abstract data structures, reactive object models, and subsystem configurations. This exercise involved extending the UML notation minimally with class stereotypes to capture generic reactive classes and port types, and an association stereotype to define communication channels between objects. Using a formal definition of the abstract reactive model as a basis, we have outlined a mapping from the UML visual model to the formal notation. The mapping supports an automatic translation mechanism from reactive object models and subsystem configurations to formal specifications. The modeling technique alleviates a designer from the need to be knowledgeable in the formal notation. The formal specifications can subsequently be used for rigorous analysis of system designs.

An integral part of the specification framework includes visual models based on minimal extensions to UML notation for capturing time-dependent properties in a design. We have incorporated these extensions within the notational convention of UML. Abstracting the basic concepts governing the structure and behavior of a reactive object, we have identified the UML model elements and extensions that are sufficient for creating the models. The resulting modeling technique provides for encapsulation of timing constraints expressed in UML statechart diagrams into generic reactive classes. We illustrated the adequacy of the technique by modeling a distributed navigation controller for road traffic. The proposed graphical notation for real-time reactive systems maps onto a succinct textual description with a simple and well-defined grammar. The semantics given in OCL applies to the UML models as well as to the textual descriptions. We have laid the foundation for a front-end tool for translating UML visual models developed using Rational Rose into specifications in the formal notation.

7.1.2 Formal Semantics of RTUML

Another aspect of our research involves the definition of formal semantics for UML real-time design models in the specification language of PVS, so as to provide a foundation for validation and verification methodologies. This process includes formalizing the syntax and well-formedness rules, and providing consistency rules for the RTUML set of modeling diagrams. The purpose of the formalization exercise is to allow precise analysis of the consequences of a design specification. For instance, analyzing the static structure diagrams of a system should allow determining whether an object is composed of certain attributes, as well as the inherence of dependencies between objects in a system and between objects across subsystems. The proposed formal semantics of RTUML serve as a sound foundation for developing a process model for object-oriented development of real-time reactive systems, and a tool to support modeling and design analysis.

The denotational semantics consists of a mapping from the set of syntactic constructs supported by RTUML to a semantic domain. This consists of formalizing the RTUML abstract syntax and well-formedness rules, providing a framework for checking the syntactic correctness of reactive system models described in the RTUML notation. It also includes defining the concepts in a semantic domain based on the abstract reactive object model. The mapping from the syntactic constructs to the semantic domain concepts is complete in the sense that every RTUML construct is mapped onto a semantic domain concept, providing a meaning for the construct. The operational semantics capture the behavior of reactive system models described in RTUML, and provide a foundation for the mechanized verification methodology.

The formal semantics serves as the basis for the development of tools supporting various stages in the development process. Such tools are necessary to support rigorous analysis of design models, such as checking whether a design specification satisfies certain properties. In addition, tools allow determining whether aspects of a design, specified using different UML diagrams, are consistent. Tools should provide for syntactic and semantic analysis, simulation by executing design specifications, analysis and debugging facilities.

7.1.3 Mechanized Verification Methodology

We have presented a methodology for verification of time-dependent properties in the design of real-time reactive systems. The correctness of a design with respect to its satisfaction of a desired property can be formally verified using an interactive theorem prover. The formal verification framework is based on PVS specification and verification environment that includes an expressive specification language and a powerful reasoning system, and is being adopted world-wide for both academic and industrial use.

We have described a technique for deriving axioms from formal design descriptions based on the RTUML object-oriented specification method. The axioms employ the relative timing of event occurrences expressed in terms of a transition time function to describe temporal relations. We have provided a justification for this methodology by relating the transition time function to the *since* operator, and the axioms based on event occurrence times to axioms based on durations derived from the logical semantics. The property to be proved in the design is specified as a theorem; a construction of the proof establishes the theorem to be a consequence of the axiomatic description. A significant aspect of this methodology is the reduction of the verification process to checking the consistency of a set of linear inequalities involving time values. We illustrated the proof steps for verifying the safety property for the generalized railroad crossing problem.

Applying the methodology to formal specifications of generic reactive classes and a configuration of instances of the classes, we demonstrate how to verify time-dependent properties in a subsystem. To formalize the verification of reactive subsystems, we have embedded the formal model in the higher-order logic of PVS. We can thus translate the specifications of generic classes and subsystems into the specification language of PVS. The dynamic aspects of a reactive system design are translated into an axiomatic description. The proof construction process is mechanized, making use of PVS theorem prover.

The distinctive feature of the proposed methodology lies in its automation. From a RTUML design of a system, a translator generates formal specifications in the form of generic reactive classes and subsystem configurations. A second translator generates PVS theories containing axiomatic descriptions of the classes and subsystems. The PVS theories maintain the object-oriented nature and modularity of the design, thereby making the theories understandable, and the methodology scalable. There are two limitations to automating the proposed verification methodology. The first is that certain axioms specific to the functionality of the system cannot be automatically generated, and need to be manually added to the theories. A second limitation is that the automatic generation of axioms may not be possible in certain designs. For instance, in the case of a reflexive transition, and in the case of two transitions labeled with the same trigger event, the axioms need to be manually introduced in the theories. However, as opposed to the first one, this limitation may be overcome by more advanced axiom generation algorithms.

The proposed verification methodology is applicable to formalisms based on state automata augmented with time and synchronous message passing, such as the Timed Automata proposed by Alur and Dill [AD96]. In an object-oriented framework, a class represents a set of objects whose computation can be defined in terms of a state machine. The behavior of each instance of the class conforms to transitions in the automaton, and to timing constraints on reactions to event occurrences in the automaton.

7.2 Future Work

This work is part of our on-going research on formal methods for real-time reactive system development, and integrating such methods with notations used in industrial practice. We have developed a validation tool supporting simulation of formal models, and formal reasoning and verification techniques for the model. Issues that are currently being investigated include

- incorporating other forms of message passing in the model for real-time reactive systems, such as parameterized events for data communication,
- developing a methodology for generating high-level source code for the design models, and
- formulating a theory for design patterns within UML notational conventions, and using the denotational semantics as foundation.

Some of the important directions for future work are outlined below.

- Adapting RTUML formal semantics to support different forms of subtyping for reactive object models.
- A study of language design issues for implementing RTUML design models, and for decoupling functionalities and real-time constraints into different layers of implementation.

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. Revised July 1998, February 1999.*

[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, OR, 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.

[ABM99] V. S. Alagar, V. Bhaskar, and D. Muthiayen. A language and environment for rigorous modeling and analysis of e-commerce architectures. Submitted for publication, October 1999.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., New York, NY, 1996.

[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.

[AD96] R. Alur and D. L. Dill. Automata-theoretic verification of real-time systems. In *Formal Methods for Real-Time Computing*, pages 55–81. John Wiley & Sons, 1996. (Eds.) C. Heitmeyer and D. Mandrioli.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

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

[AH96a] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium, RTAS'96*, Boston, MA, June 1996.

[AH96b] M. Archer and C. Heitmeyer. Tame: A specialized specification and verification system for timed automata. In *Proceedings of the 17th IEEE Real-Time Systems Symposium Work In Progress Session, RTSS'96-WIP*, Washington, DC, December 1996.

[AHH96] R. Alur, T. A. Henzinger, and P. H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[AHS98] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proceedings of User Interfaces for Theorem Provers, UITP'98*, Eindhoven, Netherlands, July 1998.

[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.

[AM99] V. S. Alagar and D. Muthiayen. A formal approach to UML modeling of complex real-time reactive systems. Submitted for publication in *IEEE Transactions on Software Engineering*, June 1999.

[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.

[AMP99a] V. S. Alagar, D. Muthiayen, and F. Pompeo. From behavioral specification to axiomatic description of real-time reactive systems. In *Proceedings of Fifth IEEE Real-Time Technology and Applications Symposium Work-in-Progress Session, RTAS'99 WIP*, Vancouver, Canada, June 1999.

[AMP99b] V. S. Alagar, D. Muthiayen, and O. Popistas. From visual modeling to formal specification of real-time reactive systems. Submitted for publication, August 1999.

[AMP99c] V. S. Alagar, D. Muthiayen, and O. Popistas. A rigorous approach for analyzing real-time UML models. Submitted for publication, October 1999.

[AMPP99] V. S. Alagar, D. Muthiayen, F. Pompeo, and O. Popistas. Verification of real-time UML models. Submitted for publication in Proceedings of Twenty-Second International Conference on Software Engineering, ICSE'2000, to be held in Limerick, Ireland, June 2000, November 1999.

[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.

[BC95] R.H. Bourdeau and B.H.C. Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, October 1995.

[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.

[BG92a] G. Berry and G. Gonthier. The esterel synchronous programming: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[BG92b] 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.

[BHH+97] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the unified modeling language. In *Proceedings of ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*, Jyvaskyla, Finland, June 1997.

[BM88] R. Boyer and J. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[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, *Proceedings of 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.

[Cal95] H. R. Callison. A time-sensitive object model for real-time systems. *ACM Transactions on Software Engineering and Methodology*, 4(3):287–317, July 1995.

[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.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching-time temporal logic. In *Proceedings of Logic of Programs Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, 1981. Springer Verlag.

[CE97] T. Clark and A. Evans. Foundations of the unified modeling language. In *Proceedings of Second Northern Formal Methods Workshop*, Ilkley, U.K., July 1997.

[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.

[Dou98a] B. P. Douglass. *Real-Time Object Orientation*. i-Logix Inc., 1998.

[Dou98b] B. P. Douglass. *Real-Time UML - Developing Efficient Objects for Embedded Systems*. Addison-Wesley, Reading, MA, 1998.

[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.

[EBF+98] A. Evans, J.-M. Bruel, R. France, Lano K., and Rumpe B. Making UML precise. In *Proceedings of OOPSLA'98 Workshop on Formalizing UML. Why? How?*, Vancouver, Canada, October 1998.

[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.

[ECM+99] A. Evans, S. Cook, S. Mellor, J. Warmer, and A. Wills. Advanced methods and tools for a precise UML. In *Proceedings of Second International Conference on the Unified Modeling Language, UML'99*, Fort Collins, CO, October 1999.

[EF97] J. Ebert and A. Fronk. Operational semantics of visual notations. Technical Report 8/97, Universitat Koblenz-Landau, Institut fur Informatik, Koblenz, Germany, 1997.

[EFLR98] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a formal modelling notation. In *Proceedings of The Unified Modeling Language - Workshop UML'98: Beyond the Notation*, Mulhouse, France, June 1998.

[ELC+98] S. Easterbrook, R. R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1):4–14, January 1998.

[FEKB98] R. France, A. Evans, Lano K., and Rumpe B. Developing the UML as a formal modeling notation. *Computer Standards and Interfaces: Special Issues on Formal Development Techniques*, 19:325–334, November 1998.

[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, ECOOP'93*, 1993.

[Gor88] M. Gordon. Mechanizing programming logics in higher-order logic. Technical Report 145, University of Cambridge, Cambridge, U.K., 1988.

[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.

[Hei98] C. Heitmeyer. On the need for practical formal methods. In *Proceedings of Fifth International Symposium on Formal Techniques in Real-Time and Real-Time Fault-Tolerant Systems, FTRTFT'98*, pages 18–26, Lyngby, Denmark, September 1998.

[HG97] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, July 1997.

[HH95] T. A. Henzinger and P. H. Ho. HYTECH: The cornell hybrid technology tool. In *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*, pages 265–293. Springer Verlag, 1995. (Eds.) P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry.

[HL74] C. A. R. Hoare and P. E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3:135–153, 1974.

[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.

[HL96] C. Heitmeyer and N. Lynch. Formal verification of real-time systems using timed automata. In *Formal Methods for Real-Time Computing*, pages 83–106. John Wiley & Sons, 1996. (Eds.) C. Heitmeyer and D. Mandrioli.

[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.

[HM96] C. Heitmeyer and B. Krishnamurthy (Series Editor) Mandrioli, D. (Eds.), editors. *Formal Methods for Real-Time Computing*. John Wiley & Sons, Inc., Chichester, U.K., 1996.

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

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

[Hoo94a] 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.

[Hoo94b] J. Hooman. Extending hoare logic to real-time. *Formal Aspects of Computing*, 6A:801–825, 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.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of Symposium on Logic in Computer Science, LICS'87*, Ithaca, NY, June 1987.

[HRdR92] J. J. M. Hooman, S. Ramesh, and W. P. de Roever. A compositional axiomatization of statecharts. *Theoretical Computer Science*, 101:289–335, 1992.

[Jac94] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, MA, 1994.

[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. Modechart: 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.

[Kel97a] P. Kellomaki. *Mechanical Verification of Invariant Properties of DisCo Specifications*. PhD thesis, Tampere University of Technology, Tampere, Finland, December 1997.

[Kel97b] P. Kellomaki. Verification of reactive systems using DisCo and PVS. In *Formal Methods Europe, FME'97*, volume 1313 of *Lecture Notes in Computer Science*, pages 589–604, Graz, Austria, September 1997. Springer-Verlag.

[KPS+93] S. Kromodimoeljo, W. Pase, M. Saaltnik, D. Craigen, and I. Meisels. A tutorial on eves. Technical report, Odyssey Research Associates, Ottawa, Ontario, 1993.

[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.

[LA94] R. R. Lutz and Y. Ampo. Experience report: Using formal methods for requirements analysis of critical spacecraft software. In *Proceedings of the Nineteenth Annual Software Engineering Workshop*, Greenbelt, MD, November 1994.

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

[Lan98] K. Lano. Defining semantics for rigorous development in UML. In *Proceedings of OOPSLA'98 Workshop on Formalizing UML. Why? How?*, Vancouver, Canada, October 1998.

[LHM+98] R. R. Lutz, G. G. Helmer, M. M. Moseman, D. E. Statezni, and S. R. Tockey. Safety analysis of requirements for a product family. In *Proceedings of the Third IEEE International Conference on Requirements Engineering, ICRE'98*, Colorado Springs, CO, April 1998.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proceedings of Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Louisiana, January 1985.

[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.

[Lut96] R. R. Lutz. Targeting safety-related errors during software requirements analysis. *The Journal of Systems and Software*, 34:223–230, September 1996.

[Lut97] R. R. Lutz. Reuse of a formal model for requirements validation. In *Proceedings of Fourth NASA Langley Formal Methods Workshop*, Hampton, VA, September 1997.

[MA98a] D. Muthiayen and V. S. Alagar. Formalizing UML for rigorous software development. In *Proceedings of Workshop on Formalizing UML. Why? How? held at Thirteenth ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'98*, Vancouver, Canada, October 1998.

[MA98b] D. Muthiayen and V. S. Alagar. A UML-based methodology for real-time reactive system development. In *Proceedings of Workshop on Behavioral Semantics of Object-Oriented Business and System Specifications held at Thirteenth ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'98*, Vancouver, Canada, October 1998.

[MA99] D. Muthiayen and V. S. Alagar. Mechanized verification of real-time reactive systems in an object-oriented framework. Submitted for publication in *IEEE Transactions on Software Engineering*, June 1999.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[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.

[MR94] S. Mauw and M. A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4), 1994.

[MSJ96] A. K. Mok, D. A. Stuart, and F. Jahanian. Specification and analysis of real-time systems: Modechart language and toolset. In *Formal Methods for Real-Time Computing*, pages 33–53. John Wiley & Sons, 1996. (Eds.) C. Heitmeyer and D. Mandrioli.

[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.

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

[Mut98] D. Muthiayen. Real-time reactive system development – a formal approach based on UML and PVS. In *Proceedings of Doctoral Symposium held at Thirteenth IEEE International Conference on Automated Software Engineering, ASE'98*, Honolulu, Hawaii, October 1998.

[NAS95] NASA. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Vol. 1: Planning and Technology Insertion*. Report NASA-GB-002-95. NASA Office of Safety and Mission Assurance, Washington D.C., 1995.

[NAS97] NASA. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Vol. 2: A Practitioner's Companion*. Report NASA-GB-001-97. NASA Office of Safety and Mission Assurance, Washington D.C., 1997.

[Obj98a] *ObjecTime Workshop on Research in Real-Time Object-Oriented Modeling*. ObjecTime Developer, January 1998.

[Obj98b] ObjecTime Limited. *UML for Real-Time*, April 1998.

[OMG99] Object Management Group, Inc. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: a prototype verification system. In *Proceedings of 11th International Conference on Automated Deduction, CADE'92*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, 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.

[OS93] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical Report CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[OS97] S. Owre and N. Shankar. The formal semantics of PVS. Technical Report CSL-97-2R, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[OSRSC99a] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS language reference, version 2.3. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[OSRSC99b] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS system guide, version 2.3. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[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.

[PAM99] K. Periyasamy, V.S. Alagar, and D. Muthiayen. Verification and validation techniques for object-oriented software systems. In *Proceedings of Thirtieth Conference on Technology of Object-Oriented Languages and Systems, TOOLS USA'99*, Santa Barbara, CA, August 1999.

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

[Pom99] F. Pompeo. A verification assistant for tromlab environment. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, September 1999.

[Pop99] O. Popistas. Rose-GRC translator: Mapping UML visual models onto formal specifications. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, April 1999.

[Rat97] Rational Software Corporation. *Unified Modeling Language for Real-Time Systems Design*, September 1997.

[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.

[RSV87] J. L. Richier, J. Sifakis, and J. Voiron. Verification in xesar of the sliding window protocol. In *Proceedings of International Symposium on Protocol Specification, Testing and Validation*, Zurich, May 1987.

[Rum98] B. Rumpe. A note on semantics (with an emphasis on UML). In *Second ECOOP Workshop on Precise Behavioral Semantics, ECOOP'98*, pages 177–197, Brussels, Belgium, July 1998.

[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.

[Sha92] N. Shankar. Mechanized verification of real-time systems using PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, November 1992.

[Sha93] 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.

[SORSC99] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS prover guide, version 2.3. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[SR98] B. Selic and J. Rumbaugh. *Using UML for Modeling Complex Real-Time Systems*, March 1998.

[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.

[WHCS97] J. Warmer, J. Hogg, S. Cook, and B. Selic. Experience with formal specification of CMM and UML. In *Proceedings of ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*, Jyvaskyla, Finland, June 1997.

[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

Mapping UML Packages to PVS Theories

In formalizing the UML metamodel, we maintain the modularity of the abstract syntax. For instance, each UML subpackage is defined in the form of (i) a PVS theory that includes the type definitions for each model element in the subpackage, (ii) a PVS theory that includes the auxiliary functions used to define the well-formedness rules of the model elements, and (iii) a PVS theory that includes the predicates and lemmas defining the well-formedness rules. In this appendix, we give a series of tables showing the corresponding PVS Theories for each UML subpackage.

A.1 UML Packages and Corresponding PVS Theories

This section gives the organization of UML packages and subpackages defining the metamodel. Table 8 including the PVS theories for each subpackage; it includes the PVS theories defining the RTUML well-formedness rules.

The following list includes the UML packages and subpackages that are used in defining RTUML notation.

1. Foundation
   (a) Core
      i. Backbone
      ii. Relationships
      iii. Dependencies
      iv. Classifiers
      v. AuxiliaryElements
   (b) ExtensionMechanisms
   (c) DataTypes
2. BehavioralElements
   (a) CommonBehavior
      i. Signals
      ii. Actions
      iii. InstancesAndLinks
   (b) Collaborations
   (c) StateMachines
   (d) UseCases

UML Package	Abstract Syntax	Auxiliary Functions	Well-Formedness Rules
Foundation			
Backbone	backbone_abs		
Relationships	relationships_abs		
Dependencies	dependencies_abs		
Classifiers	classifiers_abs		
AuxiliaryElements	auxiliaryElements_abs		
Core		core_aux	core_wfr
ExtensionMechanisms	extensionMechanisms_abs		extensionMechanisms_wfr
DataTypes	dataTypes_abs	dataTypes_aux	
BehavioralElements			
Signals	signals_abs		
Actions	actions_abs		
InstancesAndLinks	instancesAndLinks_abs		
CommonBehavior		commonBehavior_aux	commonBehavior_wfr
Collaborations	collaborations_abs	collaborations_aux	collaborations_wfr
StateMachines	stateMachines_abs	stateMachines_aux	stateMachines_wfr
UseCases	useCases_abs	useCases_aux	useCases_wfr
RTUML			
Foundation			rtumlFoundation_wfr
Collaborations			rtumlCollaborations_wfr
StateMachines			rtumlStateMachines_wfr

Table 8: PVS Theories for Formalizing UML Metamodel.

A.2 Tables Mapping UML Packages to PVS Theories

This section includes a list of tables, each enumerating the attributes, associations, and stereotypes of each model element in the subpackage, as well as the other model elements inherited by the model element. These tables are followed by a second list of tables enumerating the corresponding type definitions for the model elements, the functions defining the associations, and the auxiliary functions, and the well-formedness rules. This exercise is repeated for the UML packages *Foundation* and *Behavioral Elements*.

A.2.1 UML Package *Foundation*

Tables 9, 10, 11 and 12 list the attributes, associations, and stereotypes of each model element in the package *Foundation*. Tables 13, 14, 15, 16, 17, 18, and 19 list the corresponding type definitions for the model elements, the functions defining the associations, and the auxiliary functions, and the well-formedness rules.

UML Metaclass	Inherits	Attributes	Associations	Stereotypes
Abstraction	Dependency	mapping		derive realize refine trace
Association	Relationship GeneralizationElement	name	connection	implicit
AssociationClass	Association Class	aggregation changeability ordering isNavigable multiplicity name targetScope visibility	qualifier specification type	association global local parameter self
Attribute	StructuralFeature	changeability initialValue multiplicity targetScope	associationEnd type	
BehavioralFeature	Feature	isQuery name	parameter	create destroy
Binding	Dependency		argument	
Class	Classifier	isActive		implementationClass type
Classifier	NameSpace GeneralizableElement		feature participant powertypeRange	metaclass powerType process thread utility
Comment	ModelElement		annotatedElement	requirement responsibility
Component	Classifier		deploymentLocation resident	document executable file library table
Constraint	ModelElement	body	constrainedElement	invariant postcondition precondition
DataType	Classifier			
Dependency	Relationship		client supplier	
Element				
ElementOwnership		isSpecification visibility		
ElementResidence		visibility		

Table 9: UML Metaclasses from Package *Core*.

UML Metaclass	Inherits	Attributes	Associations	Stereotypes
Feature	ModelElement	name ownerScope visibility	owner	
Flow	Relationship			become copy
GeneralizableElement	ModelElement	isAbstract isLeaf isRoot	generalization specialization	
Generalization	Relationship	discriminator	child parent powertype	implementation
Interface	Classifier			
Method	BehavioralFeature	body	specification	
ModelElement	Element	name	clientDependency constraint implementationLocation namespace presentation supplierDependency templateParameter stereotype taggedValue	
NameSpace	ModelElement		ownedElement	
Node	Classifier		resident	
Operation	BehavioralFeature	concurrency isAbstract isLeaf isRoot		
Parameter	ModelElement	defaultValue kind name	type	
Permission	Dependency			access friend import
PresentationElement	Element			
Relationship	ModelElement			
StructuralFeature	Feature			
TemplateParameter			defaultElement	
Usage	Dependency			call create instantiate send

Table 10: UML Metaclasses from Package *Core*.

UML Metaclass	Inherits	Attributes	Associations	Stereotypes
Stereotype	GeneralizableElement	baseClass icon	extendedElement constraint requiredTag stereotypeConstraint	
TaggedValue		tag value	modelElement stereotype	

Table 11: UML Metaclasses from Package *ExtensionMechanisms*.

UML Metaclass	Inherits	Attributes	Associations	Stereotypes
ActionExpression	Expression			
AggregationKind				
ArgListsExpression	Expression			
Boolean				
BooleanExpression	Expression			
CallConcurrencyKind				
ChangeableKind				
Expression		language body		
Geometry				
Integer				
IterationExpression	Expression			
LocationReference				
Mapping		body		
MappingExpression	Expression			
MessageDirectionKind				
Multiplicity				
MultiplicityRange				
Name		body		
ObjectSetExpression	Expression			
OrderingKind				
ParameterDirectionKind				
ProcedureExpression	Expression			
PseudostateKind				
ScopeKind				
String				
Time				
TimeExpression	Expression			
TypeExpression	Expression			
UnlimitedInteger				
VisibilityKind				

Table 12: UML Metaclasses from Package *DataTypes*.

UML Package *Core – Backbone*	**PVS Theory** *backbone_abs*	**PVS Theory** *backbone_abs*	**PVS Theories** *core_aux, core_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
Element	ElementABS		
ModelElement	ModelElementABS	namespace elementOwnership constraints	suppliers allSuppliers isTemplate isInstantiated templateArguments
Feature	FeatureABS	owner	
NameSpace	NameSpaceABS		contents allContents allVisibleElements allSurroundingNamespaces WFR{1, 2}PRED
GeneralizableElement	GeneralizableElementABS		parents allParents WFR{1 .. 4}PRED
Classifier	ClassifierABS		parents allParents allFeatures allOperations allOperationsAsFeatures allMethodsAsFeatures allAttributes allAttributesAsFeatures associations allAssociations oppositeAssociationEnds allOppositeAssociationEnds specifications allContents discriminators allDiscriminators WFR{1 .. 8}PRED
StructuralFeature	StructuralFeatureABS	type	WFR{1, 2}PRED
Parameter	ParameterME	type	
BehavioralFeature	BehavioralFeatureABS		hasSameSignature matchesSignature WFR{1, 2}PRED
Constraint	ConstraintME	constrainedElements	WFR1PRED
Class ImplementationClass Type	ClassME		allContents WFR{1, 2}PRED WFR{1, 2}PRED WFR{1, 2}PRED
Attribute	AttributeME		
Operation	OperationME		
Method	MethodME	specification	WFR{1 .. 5}PRED
ElementOwnership	ElementOwnershipCC		

Table 13: Formalization of UML Package *Core – Backbone* in PVS.

UML Package *Core – Relationships*	**PVS Theory** *relationships_abs*	**PVS Theory** *relationships_abs*	**PVS Theories** *core_aux, core_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
Relationship	RelationshipABS		
Flow	FlowME	sources targets	
Generalization	GeneralizationME	child parent powertype	WFR1PRED
AssociationEnd	AssociationEndME	type specifications association	WFR{1, 2}PRED
Association	AssociationME		allConnections WFR{1 .. 4}PRED
AssociationClass	AssociationClassME		allConnections WFR{1, 2}PRED
ModelElement		sourceFlows targetFlows	
GeneralizableElement		generalizations specializations	
Classifier		powertypeRange participants associationEnds	
Attribute		associationEnd	

Table 14: Formalization of UML Package *Core – Relationships* in PVS.

UML Package *Core – Dependencies*	**PVS Theory** *dependencies_abs*	**PVS Theory** *dependencies_abs*	**PVS Theories** *core_aux, core_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
Dependency	DependencyABS	clients suppliers	
Binding	BindingME		WFR{2, 3}PRED
Abstraction	AbstractionME		
Usage	UsageME		
Permission	PermissionME		
ModelElement		clientDependencys supplierDependencys	

Table 15: Formalization of UML Package *Core – Dependencies* in PVS.

UML Package *Core – Classifiers*	**PVS Theory** *classifiers_abs*	**PVS Theory** *classifiers_abs*	**PVS Theories** *core_aux, core_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
Interface	InterfaceME		WFR{1 .. 3}PRED
DataType	DataTypeME		WFR{1, 2}PRED
Component	ComponentME	deploymentLocations	allContents allResidentElements allVisibleElements WFR{1, 2}PRED
Node	NodeME		
ElementResidence	ElementResidenceCC		
ModelElement		implementationLocations elementResidence	

Table 16: Formalization of UML Package *Core – Classifiers* in PVS.

UML Package *Core – AuxiliaryElements*	**PVS Theory** *auxiliaryElements_abs*	**PVS Theory** *auxiliaryElements_abs*	**PVS Theories** *core_aux, core_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
PresentationElement	PresentationElementABS	subjects	
Comment	CommentME	annotatedElement	
TemplateParameter	TemplateParameterCC	defaultElement	
ModelElement		presentations templateParameters	

Table 17: Formalization of UML Package *Core – AuxiliaryElements* in PVS.

UML Package *ExtensionMechanisms*	**PVS Theory** *extensionMechanisms_abs*	**PVS Theory** *extensionMechanisms_abs*	**PVS Theories** *extensionMechanisms_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
Stereotype	StereotypeME	stereotypeConstraints extendedElements	WFR{1, 2, 4}PRED
TaggedValue	TaggedValueCC		
ModelElement		stereotype taggedValues	WFR{2, 3}PRED

Table 18: Formalization of UML Package *ExtensionMechanisms* in PVS.

UML Package *DataTypes*	**PVS Theory** *dataTypes_abs*	**PVS Theory**	**PVS Theories**
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
Integer	IntegerCC		
UnlimitedInteger	UnlimitedIntegerCC		
String	StringCC		
Time	TimeCC		
AggregationKind	AggregationKindCC		
Boolean	BooleanCC		
CallConcurrencyKind	CallConcurrencyKindCC		
ChangeableKind	ChangeableKindCC		
MessageDirectionKind	MessageDirectionKindCC		
OrderingKind	OrderingKindCC		
ParameterDirectionKind	ParameterDirectionKindCC		
PseudostateKind	PseudostateKindCC		
ScopeKind	ScopeKindCC		
VisibilityKind	VisibilityKindCC		
Mapping	MappingCC		
Name	NameCC		
LocationReference	LocationReferenceCC		
MultiplicityRange	MultiplicityRangeCC		
Multiplicity	MultiplicityCC		
Geometry	GeometryCC		
Expression	ExpressionCC		
ActionExpression	ActionExpressionCC		
ArgListsExpression	ArgListsExpressionCC		
BooleanExpression	BooleanExpressionCC		
IterationExpression	IterationExpressionCC		
MappingExpression	MappingExpressionCC		
ObjectSetExpression	ObjectSetExpressionCC		
ProcedureExpression	ProcedureExpressionCC		
TimeExpression	TimeExpressionCC		
TypeExpression	TypeExpressionCC		

Table 19: Formalization of UML Package *DataTypes* in PVS.

A.2.2 UML Package *Behavioral Elements*

Tables 20, 21, 22, and 23 list the attributes, associations, and stereotypes of each model element in the package *Behavioral Elements*. Tables 24, 25, 26, 27, 28, and 29 list the corresponding type definitions for the model elements, the functions defining the associations, and the auxiliary functions, and the well-formedness rules.

UML Metaclass	Inherits	Attributes	Associations	Stereotypes
Action	ModelElement	isAsynchronous recurrence script target	actualArgument	
ActionSequence	Action		action	
Argument		value		
AttributeLink	ModelElement		value attribute	
CallAction	Action	isAsynchronous	operation	
ComponentInstance	Instance		resident	
CreateAction	Action		instantiation	
DestroyAction	Action			
DataValue	Instance		resident	
Exception	Signal		context	
Instance	ModelElement		slot linkEnd classifier	
Link	ModelElement		association connection	
LinkEnd	ModelElement		associationEnd instance qualifierValue	association global local parameter self
LinkObject	Link Object			
NodeInstance	Instance		resident	
Object	Instance			
Reception	BehavioralFeature	isAbstract isLeaf isRoot specification	signal	
ReturnAction	Action			
SendAction	Action		signal	
Signal	Classifier		context reception	
Stimulus	ModelElement		argument communicationLink dispatchAction receiver sender	
TerminateAction	Action			
UninterpretedAction	Action			

Table 20: UML Metaclasses from Package *CommonBehavior*.

UML Metaclass	Inherits	Attributes	Associations	Stereotypes
AssociationEndRole	AssociationEnd	collaborationMultiplicity	availableQualifier base	
AssociationRole	Association	multiplicity	base	
ClassifierRole	Classifier	multiplicity	availableContents availableFeature base	
Collaboration	GeneralizableElement NameSpace		constrainingElement interaction ownedElement representedClassifier representedOperation	
Interaction	ModelElement		context message	
Message	ModelElement		action activator communicationConnection interaction receiver predecessor sender	

Table 21: UML Metaclasses from Package *Collaborations*.

UML Metaclass	Inherits	Attributes	Associations	Stereotypes
Actor	Classifier		argument	
Extend	Relationship	condition	base extension extensionPoint	
ExtensionPoint	ModelElement	location		
Include	Relationship		addition base	
UseCase	Classifier		extend extensionPoint include	
UseCaseInstance	Instance			

Table 22: UML Metaclasses from Package *UseCases*.

UML Metaclass	Inherits	Attributes	Associations	Stereotypes
CallEvent	Event		operation	create destroy
ChangeEvent	Event	changeExpression		
CompositeState	State	isConcurrent isRegion	subvertex	
Event	ModelElement		parameter	
FinalState	State			
Guard	ModelElement	expression		
PseudoState	StateVertex	kind		
SignalEvent	Event		signal	
SimpleState	State			
State	StateVertex		deferrableEvent entry exit doActivity internalTransition	
StateMachine	ModelElement		context top transition	
StateVertex	ModelElement		outgoing incoming container	
StubState	StateVertex		referenceState	
SubmachineState	CompositeState		submachine	
SynchState	StateVertex	bound		
TimeEvent	Event	when		
Transition	ModelElement		trigger guard effect source target	

Table 23: UML Metaclasses from Package *StateMachines*.

UML Package	**PVS Theory**	**PVS Theory**	**PVS Theories**
CommonBehavior - Signals	*signals_abs*	*signals_abs*	*commonBehavior_aux*, *commonBehavior_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
Signal	SignalME	contexts receptions	
Exception	ExceptionME		
Reception	ReceptionME	signal	WFR1PRED
BehavioralFeature		raisedSignals	

Table 24: Formalization of UML Package *CommonBehavior - Signals* in PVS.

UML Package *CommonBehavior - Actions*	**PVS Theory** *actions_abs*	**PVS Theory** *actions_abs*	**PVS Theories** *commonBehavior_aux, commonBehavior_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
Argument	ArgumentME		
Action	ActionABS		
ActionSequence	ActionSequenceME		
CreateAction	CreateActionME	instantiation	WFR1PRED
CallAction	CallActionME	operation	WFR1PRED
ReturnAction	ReturnActionME		
SendAction	SendActionME	signal	WFR{1, 2}PRED
TerminateAction	TerminateActionME		WFR{1, 2}PRED
UninterpretedAction	UninterpretedActionME		
DestroyAction	DestroyActionME		WFR1PRED

Table 25: Formalization of UML Package *CommonBehavior - Actions* in PVS.

UML Package *CommonBehavior - InstancesAndLinks*	**PVS Theory** *instancesAndLinks_abs*	**PVS Theory** *instancesAndLinks_abs*	**PVS Theories** *commonBehavior_aux, commonBehavior_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
AttributeLink	AttributeLinkME	value	WFR1PRED
Instance	InstanceME	classifiers linkEnds	allLinks allOppositeLinkEnds selectedLinkEnds selectedAttributeLinks WFR{1 .. 6}PRED
Stimulus	StimulusME	receiver sender dispatchAction communicationLink	WFR{1, 2}PRED
LinkEnd	LinkEndME	instance associationEnd link	WFR1PRED
Link	LinkME	association	WFR{1, 2}PRED
DataValue	DataValueME		WFR{1, 2}PRED
ComponentInstance	ComponentInstanceME		WFR1PRED
NodeInstance	NodeInstanceME		WFR{1, 2}PRED
Object	ObjectME		WFR1PRED
LinkObject	LinkObjectME		WFR{1, 2}PRED
Classifier		instances	
Association		links	

Table 26: Formalization of UML Package *CommonBehavior - InstancesAndLinks* in PVS.

UML Package *Collaborations*	**PVS Theory** *collaborations_abs*	**PVS Theory** *collaborations_abs*	**PVS Theories** *collaborations_aux* *collaborations_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
ClassifierRole	ClassifierRoleME	base	allAvailableFeatures allAvailableContents WFR{1 .. 3}PRED
AssociationEndRole	AssociationEndRoleME	type base	WFR{1 .. 4}PRED
AssociationRole	AssociationRoleME	base	WFR{1, 2}PRED
Message	MessageME	communicationConnection action receiver sender predecessors activator interaction	allPredecessors WFR{1 .. 6}PRED
Interaction	InteractionME	context	WFR1PRED
Collaboration	CollaborationME	constrainingElements representedOperation representedClassifier	allContents WFR{1 .. 5}PRED

Table 27: Formalization of UML Package *Collaborations* in PVS.

UML Package *UseCases*	**PVS Theory** *useCases_abs*	**PVS Theory** *useCases_abs*	**PVS Theories** *useCases_aux* *useCases_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
ExtensionPoint	ExtensionPointME		WFR1PRED
Actor	ActorME		WFR{1, 2}PRED
UseCase	UseCaseME	includes extends extensionPoints	specificationPath allExtensionPoints WFR{1 .. 4}PRED
UseCaseInstance	UseCaseInstanceME		WFR1PRED
Include	IncludeME	addition base	
Extend	ExtendME	extension base extensionPoints	WFR1PRED

Table 28: Formalization of UML Package *UseCases* in PVS.

UML Package *StateMachines*	**PVS Theory** *stateMachines_abs*	**PVS Theory** *stateMachines_abs*	**PVS Theories** *stateMachines_aux* *stateMachines_wfr*
UML Metaclass	PVS Type Definition	Associations	Auxiliary Functions/ Well-Formedness Rules
StateVertex	StateVertexABS	outgoings incomings container	WFRPREDAUX
Event	EventABS		
Guard	GuardME		WFR1PRED
Transition	TransitionME	source target stateMachine	WFRPREDAUX WFR{1 .. 8}PRED
State	StateME	deferrableEvents	WFRPREDAUX
PseudoState	PseudoStateME		WFR{1 .. 6}PRED
SynchState	SynchStateME		WFR1PRED
StubState	StubStateME		
StateMachine	StateMachineME	context	WFRPREDAUX WFR{1 .. 5}PRED
CompositeState	CompositeStateME		WFR{1 .. 6}PRED
SimpleState	SimpleStateME		
FinalState	FinalStateME		WFR1PRED
SubmachineState	SubmachineStateME	submachine	WFR{1, 2}PRED
SignalEvent	SignalEventME	signal	
CallEvent	CallEventME	operation	
TimeEvent	TimeEventME		
ChangeEvent	ChangeEventME		
ModelElement		behaviors	
Signal		occurrences	
Operation		occurrences	

Table 29: Formalization of UML Package *StateMachines* in PVS.

A.3 PVS Theories Containing UML Stereotype Definitions

Table 30 enumerates the sets of stereotypes for each model element, as well as the PVS theory containing the definition for the set.

UML Package	UML Metaclass	PVS Type Definition	PVS Theory
Core - Backbone	BehavioralFeature	BehavioralFeatureSTES	backbone_abs
	Classifier	ClassifierSTES	backbone_abs
	Class	ClassSTES	backbone_abs
	Constraint	ConstraintSTES	backbone_abs
Core - Relationships	Flow	FlowSTES	relationships_abs
	Generalization	GeneralizationSTES	relationships_abs
	AssociationEnd	AssociationEndSTES	relationships_abs
	Association	AssociationSTES	relationships_abs
Core - Dependencies	Abstraction	AbstractionSTES	dependencies_abs
	Usage	UsageSTES	dependencies_abs
	Permission	PermissionSTES	dependencies_abs
Core - Classifiers	Component	ComponentSTES	classifiers_abs
Core - AuxiliaryElements	Comment	CommentSTES	auxiliaryElements_abs
StateMachines	CallEvent	CallEventSTES	stateMachines_abs

Table 30: PVS Type Definitions for Stereotypes of UML Metaclasses.

A.4 PVS Theories Containing RTUML Well-formedness Rules

Table 31 lists the well-formedness rules for each RTUML model element, and the PVS theory containing the predicates defining the rules.

UML Package	PVS Theory	PVS Theory
UML Metaclass	PVS Type Definition	RTUML Well-Formedness Rules
Foundation – Core – Backbone	*backbone_abs*	*rtumlFoundation_wfr*
Class	ClassME	WFR{1 .. 3}PRED
GRCClass	ClassME	WFR{1 .. 11}PRED
PortTypeClass	ClassME	WFR{1 .. 8}PRED
Foundation – Core – Relationships	*relationships_abs*	*rtumlFoundation_wfr*
Association	AssociationME	WFR{1 .. 8}PRED
PortAggregationAssociation	AssociationME	WFR1PRED
PortLinkAssociation	AssociationME	WFR1PRED
BehavioralElements – Collaborations	*collaborations_abs*	*rtumlCollaborations_wfr*
Collaboration	CollaborationME	WFR{1 .. 5}PRED
ClassifierRole	ClassifierRoleME	WFR{1 .. 2}PRED
AssociationRole	AssociationRoleME	WFR{1 .. 5}PRED
Interaction	InteractionME	WFR{1 .. 3}PRED
BehavioralElements – StateMachines	*stateMachines_abs*	*rtumlStateMachines_wfr*
StateMachine	StateMachineME	WFR1PRED
Transition	TransitionME	WFR{1 .. 7}PRED
State	StateME	WFR{1 .. 4}PRED
CompositeState	CompositeStateME	WFR{1 .. 2}PRED

Table 31: Formalization of RTUML Well-formedness Rules in PVS.

Appendix B

Formalization of UML Metamodel in PVS

The first step in defining RTUML semantics consists of formalizing the abstract syntax defining the metamodel of the core UML notation Subsequently, the stereotype extensions are introduced as new elements in the set of stereotypes for a given model element, and the RTUML well-formedness rules defined in terms of the model elements. In this appendix, we include the PVS type definitions for the model elements, the stereotype definitions, the auxiliary functions used in defining the well-formedness rules, and the definitions of the well-formedness rules in the form of predicates and lemmas. Section B.1 gives the PVS Theories for the formalization of the UML Package Foundation; Section B.2 gives the PVS Theories for the formalization of the UML Package Behavioral Elements. The new stereotypes are included in the definitions of the sets of stereotypes. The RTUML well-formedness rules are included in Appendix C.

B.1 PVS Theories for UML Package *Foundation*

B.1.1 UML Package *Datatypes*

```
dataTypes_abs: THEORY
 BEGIN

  IntegerCC: TYPE = integer

  UnlimitedIntegerCC: TYPE = integer

  UNLIMITED: UnlimitedIntegerCC

  StringCC: TYPE = string

  UNDEFINED: StringCC

  TimeCC: TYPE

  AggregationKindCC: TYPE = {none, aggregate, composite}

  BooleanCC: TYPE = boolean

  CallConcurrencyKindCC: TYPE = {sequential, guarded, concurrent}

  ChangeableKindCC: TYPE = {none, frozen, addOnly}

  MessageDirectionKindCC: TYPE = {activation, return}

  OrderingKindCC: TYPE = {unordered, ordered}

  ParameterDirectionKindCC: TYPE = {input, output, inout, return}

  PseudostateKindCC: TYPE =
  {initial, deepHistory, shallowHistory, join, fork, junction, choice}

  ScopeKindCC: TYPE = {classifier, instance}

  VisibilityKindCC: TYPE = {public, protected, private}

  MappingCC: TYPE = [# body: StringCC #]

  NameCC: TYPE = [# body: StringCC #]

  LocationReferenceCC: TYPE
```

```
  MultiplicityRangeCC: TYPE =
  [# lower: IntegerCC, upper: UnlimitedIntegerCC #]

  MultiplicityCC: TYPE =
  [# ranges: finite_set[MultiplicityRangeCC], max: UnlimitedIntegerCC #]

  GeometryCC: TYPE

  ExpressionCC: TYPE = [# language: NameCC, body: StringCC #]

  ActionExpressionCC: TYPE = [# isaExpression: ExpressionCC #]

  ArgListsExpressionCC: TYPE = [# isaExpression: ExpressionCC #]

  BooleanExpressionCC: TYPE = [# isaExpression: ExpressionCC #]

  IterationExpressionCC: TYPE = [# isaExpression: ExpressionCC #]

  MappingExpressionCC: TYPE = [# isaExpression: ExpressionCC #]

  ObjectSetExpressionCC: TYPE = [# isaExpression: ExpressionCC #]

  ProcedureExpressionCC: TYPE = [# isaExpression: ExpressionCC #]

  TimeExpressionCC: TYPE = [# isaExpression: ExpressionCC #]

  TypeExpressionCC: TYPE = [# isaExpression: ExpressionCC #]
END dataTypes_abs
```

```
dataTypes_aux : THEORY
 BEGIN

  ElementSubtypeAUX : TYPE =
  {ElementABS, ModelElementABS, FeatureABS, NameSpaceABS, GeneralizableElementABS,
    ParameterME, ConstraintME, ClassifierABS, ClassME, InterfaceME, DataTypeME,
    NodeME, ComponentME, StructuralFeatureABS, BehavioralFeatureABS, AttributeME,
    OperationME, MethodME, RelationshipABS, FlowME, GeneralizationME, AssociationME,
    AssociationEndME, AssociationClassME, DependencyABS, BindingME, AbstractionME,
    UsageME, PermissionME, SignalME, ExceptionME, ReceptionME, ArgumentME,
    ActionME, CreateActionME, CallActionME, AssignmentActionME, ReturnActionME,
    SendActionME, TerminateActionME, UninterpretedActionME, DestroyActionME,
    AttributeLinkME, StimulusME, LinkME, LinkEndME, InstanceME, DataValueME,
    ComponentInstanceME, NodeInstanceME, ObjectME, LinkObjectME, CollaborationME,
    InteractionME, MessageME, AssociationRoleME, AssociationEndRoleME, ClassifierRoleME,
    UseCaseME, ActorME, IncludeME, ExtendME, UseCaseInstanceME, ExtensionPointME,
    StateMachineME, StateVertexABS, GuardME, TransitionME, ActionABS, EventABS,
    PseudoStateME, SynchStateME, StubStateME, StateME, CompositeStateME,
    SimpleStateME, FinalStateME, SubmachineStateME, SubsystemME, SignalEventME,
    CallEventME, TimeME, ChangeME}
 END dataTypes_aux
```

B.1.2 UML Subpackage *Core – Backbone*

```
backbone_abs: THEORY
 BEGIN

  IMPORTING finite_sets@card_def, dataTypes_abs, dataTypes_aux

  ElementABS: TYPE = [# isKindOf: [ElementSubtypeAUX → boolean] #]

  ModelElementABS: TYPE = [# isaElement: ElementABS, name: NameCC #]

  FeatureABS: TYPE =
  [# isaModelElement: ModelElementABS,
     ownerScope: ScopeKindCC,
     visibility: VisibilityKindCC #]

  NameSpaceABS: TYPE =
  [# isaModelElement: ModelElementABS,
     ownedElements: finite_set[ModelElementABS] #]

  GeneralizableElementABS: TYPE =
  [# isaModelElement: ModelElementABS,
     isRoot: boolean,
     isLeaf: boolean,
     isAbstract: boolean #]

  ClassifierABS: TYPE =
  [# isaNameSpace: NameSpaceABS,
     isaGeneralizableElement: GeneralizableElementABS,
     features: finite_sequence[FeatureABS] #]

  StructuralFeatureABS: TYPE =
  [# isaFeature: FeatureABS,
     multiplicity: MultiplicityCC,
     changeability: ChangeableKindCC,
     targetScope: ScopeKindCC #]

  ParameterME: TYPE =
  [# isaModelElement: ModelElementABS,
     defaultValue: ExpressionCC,
     kind: ParameterDirectionKindCC #]

  BehavioralFeatureABS: TYPE =
  [# isaFeature: FeatureABS,
     isQuery: boolean,
```

```
    parameters: finite_sequence[ParameterME] #]

ConstraintME: TYPE =
[# isaModelElement: ModelElementABS, body: BooleanExpressionCC #]

ClassME: TYPE = [# isaClassifier: ClassifierABS, isActive: boolean #]

AttributeME: TYPE =
[# isaStructuralFeature: StructuralFeatureABS,
    initialValue: ExpressionCC #]

OperationME: TYPE =
[# isaBehavioralFeature: BehavioralFeatureABS,
    concurrency: CallConcurrencyKindCC,
    isRoot: boolean,
    isLeaf: boolean,
    isAbstract: boolean,
    specification: StringCC #]

MethodME: TYPE =
[# isaBehavioralFeature: BehavioralFeatureABS,
    body: ProcedureExpressionCC #]

ElementOwnershipCC: TYPE =
[# visibility: VisibilityKindCC, isSpecification: boolean #]

elementOwnershipASS: [ModelElementABS → ElementOwnershipCC]

namespaceASS: [ModelElementABS → NameSpaceABS]

constraintsASS: [ModelElementABS → finite_set[ConstraintME]]

ownerASS: [FeatureABS → ClassifierABS]

typeASS: [StructuralFeatureABS → ClassifierABS]

typeASS: [ParameterME → ClassifierABS]

constrainedElementsASS:
[ConstraintME → finite_sequence[ModelElementABS]]

specificationASS: [MethodME → OperationME]

BehavioralFeatureSTES: TYPE = {createSTE, destroySTE}
```

```
ClassifierSTES: TYPE =
{metaclassSTE, powertypeSTE, processSTE, threadSTE, utilitySTE}

ClassSTES: TYPE =
{implementationClassSTE, typeSTE, grcSTE, porttypeSTE}

ConstraintSTES: TYPE =
{invariantSTE, postconditionSTE, preconditionSTE}

stereotypeAUX: [BehavioralFeatureABS → BehavioralFeatureSTES]

stereotypeAUX: [ClassifierABS → ClassifierSTES]

stereotypeAUX: [ClassME → ClassSTES]

stereotypeAUX: [ConstraintME → ConstraintSTES]
END backbone_abs
```

B.1.3 UML Subpackage *Core – Relationships*

```
relationships_abs: THEORY
 BEGIN

  IMPORTING backbone_abs

  RelationshipABS: TYPE = [# isaModelElement: ModelElementABS #]

  FlowME: TYPE = [# isaRelationship: RelationshipABS #]

  GeneralizationME: TYPE =
  [# isaRelationship: RelationshipABS, discriminator: NameCC #]

  AssociationEndME: TYPE =
  [# isaModelElement: ModelElementABS,
     isNavigable: boolean,
     ordering: OrderingKindCC,
     aggregation: AggregationKindCC,
     targetScope: ScopeKindCC,
     multiplicity: MultiplicityCC,
     changeability: ChangeableKindCC,
     visibility: VisibilityKindCC,
     qualifiers: finite_sequence[AttributeME] #]

  AssociationME: TYPE =
  [# isaRelationship: RelationshipABS,
     isaGeneralizableElement: GeneralizableElementABS,
     connections: finite_sequence[AssociationEndME] #]

  AssociationClassME: TYPE =
  [# isaClass: ClassME, isaAssociation: AssociationME #]

  sourcesASS: [FlowME → finite_set[ModelElementABS]]

  targetsASS: [FlowME → finite_set[ModelElementABS]]

  childASS: [GeneralizationME → GeneralizableElementABS]

  parentASS: [GeneralizationME → GeneralizableElementABS]

  powertypeASS: [GeneralizationME → ClassifierABS]

  typeASS: [AssociationEndME → ClassifierABS]
```

specificationsASS : [AssociationEndME → finite_set[ClassifierABS]]

associationASS : [AssociationEndME → AssociationME]

sourceFlowsASS : [ModelElementABS → finite_set[FlowME]]

targetFlowsASS : [ModelElementABS → finite_set[FlowME]]

generalizationsASS :
[GeneralizableElementABS → finite_set[GeneralizationME]]

specializationsASS :
[GeneralizableElementABS → finite_set[GeneralizationME]]

powertypeRangeASS : [ClassifierABS → finite_set[GeneralizationME]]

participantsASS : [ClassifierABS → finite_set[AssociationEndME]]

associationEndsASS : [ClassifierABS → finite_set[AssociationEndME]]

associationEndASS : [AttributeME → AssociationEndME]

FlowSTES : TYPE = {becomeSTE, copySTE}

GeneralizationSTES : TYPE = {implementationSTE}

AssociationEndSTES : TYPE =
{associationSTE, globalSTE, localSTE, parameterSTE, selfSTE}

AssociationSTES : TYPE = {implicitSTE, portaggregationSTE, portlinkSTE}

stereotypeAUX : [FlowME → FlowSTES]

stereotypeAUX : [GeneralizationME → GeneralizationSTES]

stereotypeAUX : [AssociationEndME → AssociationEndSTES]

stereotypeAUX : [AssociationME → AssociationSTES]
END relationships_abs

B.1.4 UML Subpackage *Core – Dependencies*

```
dependencies_abs: THEORY
 BEGIN

  IMPORTING backbone_abs, relationships_abs

  DependencyABS: TYPE = [# isaRelationship: RelationshipABS #]

  BindingME: TYPE =
  [# isaDependency: DependencyABS,
     arguments: finite_sequence[ModelElementABS] #]

  AbstractionME: TYPE =
  [# isaDependency: DependencyABS, mapping: MappingExpressionCC #]

  UsageME: TYPE = [# isaDependency: DependencyABS #]

  PermissionME: TYPE = [# isaDependency: DependencyABS #]

  clientsASS: [DependencyABS -> finite_set[ModelElementABS]]

  suppliersASS: [DependencyABS -> finite_set[ModelElementABS]]

  clientDependencysASS: [ModelElementABS -> finite_set[DependencyABS]]

  supplierDependencysASS: [ModelElementABS -> finite_set[DependencyABS]]

  AbstractionSTES: TYPE = {deriveSTE, realizeSTE, refineSTE, traceSTE}

  UsageSTES: TYPE = {callSTE, createSTE, instantiateSTE, sendSTE}

  PermissionSTES: TYPE = {accessSTE, friendSTE, importSTE}

  stereotypeAUX: [AbstractionME -> AbstractionSTES]

  stereotypeAUX: [UsageME -> UsageSTES]

  stereotypeAUX: [PermissionME -> PermissionSTES]
 END dependencies_abs
```

B.1.5 UML Subpackage *Core – Classifiers*

```
classifiers_abs: THEORY
 BEGIN

  IMPORTING backbone_abs

  InterfaceME: TYPE = [# isaClassifier: ClassifierABS #]

  DataTypeME: TYPE = [# isaClassifier: ClassifierABS #]

  ComponentME: TYPE =
  [# isaClassifier: ClassifierABS,
     residents: finite_set[ModelElementABS] #]

  NodeME: TYPE =
  [# isaClassifier: ClassifierABS, residents: finite_set[ComponentME] #]

  ElementResidenceCC: TYPE = [# visibility: VisibilityKindCC #]

  elementResidenceASS: [ModelElementABS -> ElementResidenceCC]

  implementationLocationsASS: [ModelElementABS -> finite_set[ComponentME]]

  deploymentLocationsASS: [ComponentME -> finite_set[NodeME]]

  ComponentSTES: TYPE =
  {documentSTE, executableSTE, fileSTE, librarySTE, tableSTE}

  stereotypeAUX: [ComponentME -> ComponentSTES]
 END classifiers_abs
```

B.1.6 UML Subpackage *Core – Auxiliary Elements*

```
auxiliaryElements_abs: THEORY
 BEGIN

  IMPORTING backbone_abs, dependencies_abs

  PresentationElementABS: TYPE = [# isaElement: ElementABS #]

  CommentME: TYPE = [# isaModelElement: ModelElementABS #]

  TemplateParameterCC: TYPE

  subjectsASS: [PresentationElementABS → finite_set[ModelElementABS]]

  annotatedElementASS: [CommentME → finite_set[ModelElementABS]]

  defaultElementASS: [TemplateParameterCC → ModelElementABS]

  presentationsASS: [ModelElementABS → finite_set[PresentationElementABS]]

  templateParametersASS:
  [ModelElementABS → finite_sequence[ModelElementABS]]

  CommentSTES: TYPE = {requirementSTE, responsibilitySTE}

  stereotypeAUX: [CommentME → CommentSTES]
 END auxiliaryElements_abs
```

B.1.7 UML Package *Core*

```
core_aux: THEORY
 BEGIN

  IMPORTING backbone_abs, relationships_abs, dependencies_abs, classifiers_abs,
            auxiliaryElements_abs

  inheritanceHierarchyDepthAUX: [ClassifierABS → nat]

  inheritanceHierarchyDepthAUX: [NameSpaceABS → nat]

  inheritanceHierarchyDepthAUX: [GeneralizableElementABS → nat]

  supplierHierarchyDepthAUX: [ModelElementABS → nat]

  parentsAUX(c): finite_set[ClassifierABS] =
      {c2: ClassifierABS |
           ∃ (g: GeneralizationME):
             (g ∈ generalizationsASS(isaGeneralizableElement(c))) ∧
              isaGeneralizableElement(c2) = parentASS(g)}

  allParentsAUX(c: ClassifierABS): RECURSIVE finite_set[ClassifierABS] =
    LET s1: finite_set[ClassifierABS] = parentsAUX(c),
        s2: finite_set[ClassifierABS]
          = {c2: ClassifierABS |
                  ∃ (c1: ClassifierABS): (c1 ∈ s1) ∧ (c2 ∈ allParentsAUX(c1))}
     IN (s1 ∪ s2)
    MEASURE inheritanceHierarchyDepthAUX(c)

  allConnectionsAUX(a: AssociationME): finite_set[AssociationEndME] =
      {e: AssociationEndME | (e ∈ finseq2list(connections(a)))}

  allConnectionsAUX(a: AssociationClassME): finite_set[AssociationEndME] =
      LET s1: finite_set[ClassifierABS]
              = {c: ClassifierABS |
                     (c ∈ allParentsAUX(isaClassifier(isaClass(a)))) ∧
                      isKindOf(isaElement(isaModelElement(isaGeneralizableElement(c))))
                          (AssociationME)},
          s2: finite_set[AssociationClassME]
              = {a2: AssociationClassME | (isaClassifier(isaClass(a2)) ∈ s1)},
          s3: finite_set[AssociationEndME]
              = {e1: AssociationEndME | (e1 ∈ finseq2list(connections(isaAssociation(a))))},
          s4: finite_set[AssociationEndME]
              = {e2: AssociationEndME |
```

$\exists\ (a_3:\ \text{AssociationClassME}):$
$(a_3 \in s_2)\ \wedge\ (e_2 \in \text{finseq2list(connections(isAssociation}(a_3))))\}$
IN $(s_3 \cup s_4)$

hasSameSignatureAUX (b_1, b_2: BehavioralFeatureABS) : boolean =
name (isModelElement (isFeature (b_1))) = name (isModelElement (isFeature (b_2))) $\wedge$
length (parameters (b_1)) = length (parameters (b_2)) $\wedge$
($\forall$ (i: below (length (parameters (b_1)))) :
typeASS (nth (finseq2list (parameters (b_1)) , i)) =
typeASS (nth (finseq2list (parameters (b_2)) , i))
$\wedge$
kind (nth (finseq2list (parameters (b_1)) , i)) =
kind (nth (finseq2list (parameters (b_2)) , i)))

matchesSignatureAUX (b_1, b_2: BehavioralFeatureABS) : boolean =
name (isModelElement (isFeature (b_1))) = name (isModelElement (isFeature (b_2))) $\wedge$
length (parameters (b_1)) = length (parameters (b_2)) $\wedge$
($\forall$ (i: below (length (parameters (b_1)))) :
typeASS (nth (finseq2list (parameters (b_1)) , i)) =
typeASS (nth (finseq2list (parameters (b_2)) , i))
$\vee$
(kind (nth (finseq2list (parameters (b_1)) , i)) = return $\wedge$
kind (nth (finseq2list (parameters (b_2)) , i)) = return))

allContentsAUX (c: ClassME) : finite_set [ModelElementABS] =
ownedElements (isNameSpace (isClassifier (c)))

allFeaturesAUX (c: ClassifierABS) : finite_set [FeatureABS] =
LET s_1: finite_set [FeatureABS] = $\{f:$ FeatureABS $\mid$ $(f \in \text{finseq2list(features}(c)))\}$,
s_2: finite_set [FeatureABS]
= $\{f:$ FeatureABS $\mid$
$\exists\ (c_2:\ \text{ClassifierABS}):$
$(c_2 \in \text{allParentsAUX}(c))\ \wedge\ (f \in \text{finseq2list(features}(c_2)))\}$
IN $(s_1 \cup s_2)$

allOperationsAUX (c: ClassifierABS) : finite_set [OperationME] =
$\{p:$ OperationME $\mid$
$(\text{isFeature(isBehavioralFeature}(p)) \in \text{allFeaturesAUX}(c))\}$

allOperationsAsFeaturesAUX (c: ClassifierABS) : finite_set [FeatureABS] =
$\{f:$ FeatureABS $\mid$
$(f \in \text{allFeaturesAUX}(c))$ $\wedge$
isKindOf (isElement (isModelElement (f))) (OperationME) $\}$

allMethodsAsFeaturesAUX (c: ClassifierABS) : finite_set [FeatureABS] =

```
    {f: FeatureABS |
        (f ∈ allFeaturesAUX(c)) ∧
         isKindOf(isaElement(isaModelElement(f)))(MethodME)}

allAttributesAUX(c: ClassifierABS): finite_set[AttributeME] =
    {a: AttributeME |
        (isaFeature(isaStructuralFeature(a)) ∈ allFeaturesAUX(c))}

allAttributesAsFeaturesAUX(c: ClassifierABS): finite_set[FeatureABS] =
    {f: FeatureABS |
        (f ∈ allFeaturesAUX(c)) ∧
         isKindOf(isaElement(isaModelElement(f)))(AttributeME)}

associationsAUX(c: ClassifierABS): finite_set[AssociationME] =
    {a: AssociationME |
        ∃ (ae: AssociationEndME):
          (ae ∈ associationEndsASS(c)) ∧ a = associationASS(ae)}

allAssociationsAUX(c: ClassifierABS): finite_set[AssociationME] =
    LET s_1: finite_set[AssociationME] = associationsAUX(c),
        s_2: finite_set[AssociationME]
            = {a: AssociationME |
                   ∃ (c_2: ClassifierABS):
                     (c_2 ∈ allParentsAUX(c)) ∧ (a ∈ associationsAUX(c_2))}
      IN (s_1 ∪ s_2)

oppositeAssociationEndsAUX(c: ClassifierABS): finite_set[AssociationEndME] =
    LET s_1: finite_set[AssociationME]
            = {a: AssociationME |
                   (a ∈ associationsAUX(c)) ∧
                    card({e: AssociationEndME |
                              (e ∈ finseq2list(connections(a))) ∧ typeASS(e) = c})
                     = 1},
        s_2: finite_set[AssociationME]
            = {a: AssociationME |
                   (a ∈ associationsAUX(c)) ∧
                    card({e: AssociationEndME |
                              (e ∈ finseq2list(connections(a))) ∧ typeASS(e) = c})
                     > 1},
        s_3: finite_set[AssociationEndME]
            = {e: AssociationEndME |
                   ∃ (a: AssociationME):
                     (a ∈ s_1) ∧ (e ∈ finseq2list(connections(a))) ∧ typeASS(e) ≠ c},
        s_4: finite_set[AssociationEndME]
            = {e: AssociationEndME |
```

$\exists$ (a: AssociationME) : ($a \in s_2$) $\wedge$ ($e \in$ finseq2list(connections(a))) }
IN ($s_3 \cup s_4$)

allOppositeAssociationEndsAUX (c: ClassifierABS) : finite_set [AssociationEndME] =
LET s_1 : finite_set [AssociationEndME] = oppositeAssociationEndsAUX (c) ,
s_2 : finite_set [AssociationEndME]
= { e: AssociationEndME |
$\exists$ (c_2: ClassifierABS) :
($c_2 \in$ allParentsAUX(c)) $\wedge$ ($e \in$ oppositeAssociationEndsAUX(c_2)) }
IN ($s_1 \cup s_2$)

specificationsAUX (c: ClassifierABS) : finite_set [ClassifierABS] =
LET s: finite_set [DependencyABS]
= { d: DependencyABS |
($d \in$ clientDependencysASS(isaModelElement(isaGeneralizableElement(c)))) $\wedge$
isKindOf (isaElement (isaModelElement (isaRelationship (d)))) (AbstractionME) $\wedge$
($\exists$ (a: AbstractionME) : isaDependency (a) = d $\wedge$
stereotypeAUX (a) = realizeSTE) $\wedge$
($\forall$ (me : ModelElementABS) :
(me $\in$ suppliersASS(d)) $\supset$ isKindOf (isaElement (me)) (ClassifierABS)) }
IN
{ c_2 : ClassifierABS |
$\exists$ (d: DependencyABS) :
($d \in s$) $\wedge$
(isaModelElement(isaGeneralizableElement(c_2)) $\in$ suppliersASS(d)) }

allContentsAUX (c: ClassifierABS) : finite_set [ModelElementABS] =
LET s_1 : finite_set [ModelElementABS] = ownedElements (isaNameSpace (c)) ,
s_2 : finite_set [ModelElementABS]
= { m: ModelElementABS |
$\exists$ (c_2: ClassifierABS) :
($c_2 \in$ allParentsAUX(c)) $\wedge$
($m \in$ ownedElements(isaNameSpace(c_2))) $\wedge$
(visibility (elementOwnershipASS (m)) = public $\vee$
visibility (elementOwnershipASS (m)) = protected) }
IN ($s_1 \cup s_2$)

discriminatorsAUX (c: ClassifierABS) : finite_set [NameCC] =
{ n: NameCC |
$\exists$ (g: GeneralizationME) :
($g \in$ powertypeRangeASS(c)) $\wedge$ n = discriminator (g) }

allDiscriminatorsAUX (c: ClassifierABS) : finite_set [NameCC] =
LET s_1 : finite_set [NameCC] = discriminatorsAUX (c) ,
s_2 : finite_set [NameCC]

```
        = {n: NameCC |
              ∃ (c2: ClassifierABS) :
                (c2 ∈ allParentsAUX(c)) ∧ (n ∈ discriminatorsAUX(c2))}
    IN (s1 ∪ s2)

allContentsAUX(c: ComponentME) : finite_set[ModelElementABS] =
    ownedElements(isaNameSpace(isaClassifier(c)))

allResidentElementsAUX(c: ComponentME) : finite_set[ModelElementABS] =
    LET s1 : finite_set[ModelElementABS] = residents(c),
        s2 : finite_set[ModelElementABS]
          = {m: ModelElementABS |
              (∃ (c: ClassifierABS, c2: ComponentME) :
                (c ∈ allParentsAUX(isaClassifier(c))) ∧
                 isaClassifier(c2) = c ∧ (m ∈ residents(c2)))
              ∧
              (visibility(elementResidenceASS(m)) = public ∨
                visibility(elementResidenceASS(m)) = protected)}
    IN (s1 ∪ s2)

allVisibleElementsAUX(c: ComponentME) : finite_set[ModelElementABS] =
    LET s1 : finite_set[ModelElementABS]
          = {m: ModelElementABS |
              (m ∈ allContentsAUX(c)) ∧
               visibility(elementOwnershipASS(m)) = public},
        s2 : finite_set[ModelElementABS]
          = {m: ModelElementABS |
              (m ∈ allResidentElementsAUX(c)) ∧
               visibility(elementResidenceASS(m)) = public}
    IN (s1 ∪ s2)

parentsAUX(g) : finite_set[GeneralizableElementABS] =
    {g2: GeneralizableElementABS |
        ∃ (z: GeneralizationME) :
          (z ∈ generalizationsASS(g)) ∧ g2 = parentASS(z)}

allParentsAUX(g: GeneralizableElementABS) : RECURSIVE finite_set[GeneralizableElementABS] =
  LET s1 : finite_set[GeneralizableElementABS] = parentsAUX(g),
      s2 : finite_set[GeneralizableElementABS]
        = {g2: GeneralizableElementABS |
             ∃ (g1: GeneralizableElementABS) : (g1 ∈ s1) ∧ (g2 ∈ allParentsAUX(g1))}
  IN (s1 ∪ s2)
  MEASURE inheritanceHierarchyDepthAUX(g)

suppliersAUX(m) : finite_set[ModelElementABS] =
```

```
{m₂ : ModelElementABS |
      ∃ (d : DependencyABS) :
        (d ∈ clientDependencysASS(m)) ∧
         (m₂ ∈ suppliersASS(d))}

allSuppliersAUX(m : ModelElementABS) : RECURSIVE finite_set[ModelElementABS] =
  LET s₁ : finite_set[ModelElementABS] = suppliersAUX(m),
      s₂ : finite_set[ModelElementABS]
         = {m₂ : ModelElementABS |
                  ∃ (m₁ : ModelElementABS) : (m₁ ∈ s₁) ∧ (m₂ ∈ allSuppliersAUX(m₁))}
    IN (s₁ ∪ s₂)
   MEASURE supplierHierarchyDepthAUX(m)

isTemplateAUX(m : ModelElementABS) : boolean =
    ¬ length(templateParametersASS(m)) = 0

isInstantiated(m : ModelElementABS) : boolean =
    LET s : finite_set[DependencyABS]
            = {d : DependencyABS |
                   (d ∈ clientDependencysASS(m)) ∧
                    isKindOf(isaElement(isaModelElement(isaRelationship(d))))(BindingME)}
      IN ¬ empty?(s)

templateArgumentsAUX(m : ModelElementABS) : finite_set[ModelElementABS] =
    LET s : finite_set[DependencyABS]
            = {d : DependencyABS |
                   (d ∈ clientDependencysASS(m)) ∧
                    isKindOf(isaElement(isaModelElement(isaRelationship(d))))(BindingME)}
      IN
      {m₂ : ModelElementABS |
           ∃ (b : BindingME) :
             (isaDependency(b) ∈ s) ∧
              (m₂ ∈ finseq2list(arguments(b)))}

contentsAUX(n : NameSpaceABS) : RECURSIVE finite_set[ModelElementABS] =
  LET s : finite_set[ModelElementABS]
         = {m : ModelElementABS | (m ∈ contentsAUX(namespaceASS(isaModelElement(n))))}
    IN (ownedElements(n) ∪ s)
   MEASURE inheritanceHierarchyDepthAUX(n)

allContentsAUX(n : NameSpaceABS) : finite_set[ModelElementABS] =
    contentsAUX(n)

allVisibleElementsAUX(n : NameSpaceABS) : finite_set[ModelElementABS] =
    {m : ModelElementABS |
```

```
        (m ∈ allContentsAUX(n)) ∧
         visibility(elementOwnershipASS(m)) = public}

allSurroundingNamespacesAUX(n: NameSpaceABS): RECURSIVE finite_set[NameSpaceABS] =
  LET s1: finite_set[NameSpaceABS]
          = {n1: NameSpaceABS | n1 = namespaceASS(isaModelElement(n))},
      s2: finite_set[NameSpaceABS]
          = {n2: NameSpaceABS |
                (∃ (n3: NameSpaceABS):
                   (n3 ∈ s1) ∧ (n2 ∈ allSurroundingNamespacesAUX(n3)))}
   IN (s1 ∪ s2)
  MEASURE inheritanceHierarchyDepthAUX(n)
END core_aux
```

```
core_wfr: THEORY
 BEGIN

  IMPORTING instancesAndLinks_abs, extensionMechanisms_abs, core_aux

  AssociationWFR1PRED(a: AssociationME): boolean =
      (∀ (e1, e2: AssociationEndME):
         (e1 ∈ allConnectionsAUX(a)) ∧
          (e2 ∈ allConnectionsAUX(a)) ∧
           name(isModelElement(e1)) = name(isModelElement(e2))
           ⊃ e1 = e2)

  AssociationWFR2PRED(a: AssociationME): boolean =
      LET s: finite_set[AssociationEndME]
              = {e: AssociationEndME |
                     (e ∈ allConnectionsAUX(a)) ∧
                      (aggregation(e) = aggregate ∨ aggregation(e) = composite)}
        IN card(s) ≤ 1

  AssociationWFR3PRED(a: AssociationME): boolean =
      card(allConnectionsAUX(a)) ≥ 3 ⊃
       (∀ (e: AssociationEndME):
          (e ∈ allConnectionsAUX(a)) ⊃ aggregation(e) = none)

  AssociationWFR4PRED(a: AssociationME): boolean =
      LET n: NameSpaceABS = namespaceASS(isModelElement(isRelationship(a))),
          s: finite_set[ModelElementABS]
              = {m: ModelElementABS |
                     ∃ (e: AssociationEndME):
                       (e ∈ allConnectionsAUX(a)) ∧
                        isModelElement(isGeneralizableElement(typeASS(e))) = m}
        IN (s ⊆ allContentsAUX(n))

  AssociationWFRLEMMA: LEMMA
    (∀ (a: AssociationME):
       AssociationWFR1PRED(a) ∧
        AssociationWFR2PRED(a) ∧
         AssociationWFR3PRED(a) ∧ AssociationWFR4PRED(a))

  AssociationClassWFR1PRED(a: AssociationClassME): boolean =
      (∀ (e: AssociationEndME, f: FeatureABS):
         (e ∈ allConnectionsAUX(a)) ∧
          (f ∈ allFeaturesAUX(isClassifier(isClass(a)))) ∧
           isKindOf(isElement(isModelElement(f)))(StructuralFeatureABS)
          ⊃ name(isModelElement(e)) ≠ name(isModelElement(f)))
```

```
AssociationClassWFR2PRED(a: AssociationClassME): boolean =
    (∀ (e: AssociationEndME):
        (e ∈ allConnectionsAUX(a)) ⊃
         typeASS(e) ≠ isClassifier(isClass(a)))

AssociationClassWFRLEMMA: LEMMA
  (∀ (a: AssociationClassME):
     AssociationClassWFR1PRED(a) ∧ AssociationClassWFR2PRED(a))

AssociationEndWFR1PRED(e: AssociationEndME): boolean =
    LET e: ElementABS
            = isaElement(isaModelElement(isaGeneralizableElement(typeASS(e))))
      IN
      (isKindOf(e)(InterfaceME) ∨ isKindOf(e)(DataTypeME)) ⊃
       (∀ (e2: AssociationEndME):
          ((e2 ∈ finseq2list(connections(associationASS(e)))) ∧ e ≠ e2) ⊃
           ¬ isNavigable(e2))

AssociationEndWFR2PRED(e: AssociationEndME): boolean =
    aggregation(e) = composite ⊃ max(multiplicity(e)) ≤ 1

AssociationEndWFRLEMMA: LEMMA
  (∀ (e: AssociationEndME):
     AssociationEndWFR1PRED(e) ∧ AssociationEndWFR2PRED(e))

BehavioralFeatureWFR1PRED(b: BehavioralFeatureABS): boolean =
    (∀ (p1, p2: ParameterME):
        (p1 ∈ finseq2list(parameters(b))) ∧
         (p2 ∈ finseq2list(parameters(b))) ∧
          name(isaModelElement(p1)) = name(isaModelElement(p2))
         ⊃ p1 = p2)

BehavioralFeatureWFR2PRED(b: BehavioralFeatureABS): boolean =
    LET n: NameSpaceABS
          = namespaceASS(isaModelElement(isaGeneralizableElement(ownerASS(isaFeature(b)))))
      IN
      (∀ (p: ParameterME):
          (p ∈ finseq2list(parameters(b))) ⊃
           (isaModelElement(p) ∈ allContentsAUX(n)))

BehavioralFeatureWFRLEMMA: LEMMA
  (∀ (b: BehavioralFeatureABS):
     BehavioralFeatureWFR1PRED(b) ∧ BehavioralFeatureWFR2PRED(b))
```

```
BindingWFR2PRED(b: BindingME): boolean =
     card(clientsASS(isDependency(b))) = 1 ∧
      card(suppliersASS(isDependency(b))) = 1

BindingWFR3PRED(b: BindingME): boolean =
     (∀ (b2: BindingME):
        b ≠ b2 ⊃
         clientsASS(isDependency(b)) ≠ clientsASS(isDependency(b2)))

BindingWFRLEMMA: LEMMA
   (∀ (b: BindingME): BindingWFR2PRED(b) ∧ BindingWFR3PRED(b))

ClassWFR1PRED(c: ClassME): boolean =
     ¬ isAbstract(isGeneralizableElement(isClassifier(c))) ⊃
      (∀ (p: OperationME):
         (isFeature(isBehavioralFeature(p)) ∈ allOperationsAsFeaturesAUX(isClassifier(c)))
          ⊃
          (∃ (m: MethodME):
             (isFeature(isBehavioralFeature(m)) ∈ allMethodsAsFeaturesAUX(isClassifier(c)))
              ∧ specificationASS(m) = p))

ClassWFR2PRED(c: ClassME): boolean =
     (∀ (m: ModelElementABS):
        (m ∈ allContentsAUX(c)) ⊃
         isKindOf(isElement(m))(ClassME) ∨
          isKindOf(isElement(m))(AssociationME) ∨
           isKindOf(isElement(m))(GeneralizationME) ∨
            isKindOf(isElement(m))(UseCaseME) ∨
             isKindOf(isElement(m))(ConstraintME) ∨
              isKindOf(isElement(m))(DependencyABS) ∨
               isKindOf(isElement(m))(CollaborationME) ∨
                isKindOf(isElement(m))(DataTypeME) ∨
                 isKindOf(isElement(m))(InterfaceME))

ClassWFRLEMMA: LEMMA
   (∀ (c: ClassME): ClassWFR1PRED(c) ∧ ClassWFR2PRED(c))

ClassifierWFR1PRED(c: ClassifierABS): boolean =
     (∀ (b1, b2: BehavioralFeatureABS):
        (isFeature(b1) ∈ finseq2list(features(c))) ∧
         (isFeature(b2) ∈ finseq2list(features(c))) ∧
          ((isKindOf(isElement(isModelElement(isFeature(b1))))(OperationME) ∧
             isKindOf(isElement(isModelElement(isFeature(b2))))(OperationME))
            ∨
            (isKindOf(isElement(isModelElement(isFeature(b1))))(MethodME) ∧
```

isKindOf (isaElement (isaModelElement (isaFeature (b_2)))) (MethodME))
$\vee$
(isKindOf (isaElement (isaModelElement (isaFeature (b_1)))) (ReceptionME) $\wedge$
isKindOf (isaElement (isaModelElement (isaFeature (b_2)))) (ReceptionME)))
$\wedge$ matchesSignatureAUX (b_1, b_2)
$\supset$ $b_1 = b_2$)

ClassifierWFR2PRED (c: ClassifierABS) : boolean =
($\forall$ (f_1, f_2: FeatureABS) :
($f_1 \in$ finseq2list(features(c))) $\wedge$
($f_2 \in$ finseq2list(features(c))) $\wedge$
isKindOf (isaElement (isaModelElement (f_1))) (AttributeME) $\wedge$
isKindOf (isaElement (isaModelElement (f_2))) (AttributeME) $\wedge$
name (isaModelElement (f_1)) = name (isaModelElement (f_2))
$\supset$ $f_1 = f_2$)

ClassifierWFR3PRED (c: ClassifierABS) : boolean =
($\forall$ (e_1, e_2: AssociationEndME) :
($e_1 \in$ oppositeAssociationEndsAUX(c)) $\wedge$
($e_2 \in$ oppositeAssociationEndsAUX(c)) $\wedge$
name (isaModelElement (e_1)) = name (isaModelElement (e_2))
$\supset$ $e_1 = e_2$)

ClassifierWFR4PRED (c: ClassifierABS) : boolean =
LET s_1 : finite_set [FeatureABS]
= { f: FeatureABS |
($f \in$ finseq2list(features(c))) $\wedge$
isKindOf (isaElement (isaModelElement (f))) (AttributeME) },
s_2 : finite_set [AssociationEndME] = allOppositeAssociationEndsAUX (c) ,
s_3 : finite_set [ModelElementABS] = allContentsAUX (isaNameSpace (c)) ,
s_4 : finite_set [NameCC]
= { n: NameCC |
($\exists$ (e: AssociationEndME) : ($e \in s_2$) $\wedge$
name (isaModelElement (e)) = n) },
s_5 : finite_set [NameCC]
= { n: NameCC | ($\exists$ (m: ModelElementABS) : ($m \in s_3$) $\wedge$ name (m) = n) }
IN
($\forall$ (af: FeatureABS) :
(af $\in s_1$) $\supset$ $\neg$ (name(isaModelElement(af)) $\in (s_4 \cup s_5)$))

ClassifierWFR5PRED (c: ClassifierABS) : boolean =
LET s_1 : finite_set [AssociationEndME] = oppositeAssociationEndsAUX (c) ,
s_2 : finite_set [FeatureABS] = allAttributesAsFeaturesAUX (c) ,
s_3 : finite_set [ModelElementABS] = allContentsAUX (isaNameSpace (c)) ,
s_4 : finite_set [NameCC]

```
        = {n: NameCC |
               (∃ (f: FeatureABS): (f ∈ s_2) ∧ name(isModelElement(f)) = n)},
    s_5: finite_set[NameCC]
        = {n: NameCC | (∃ (m: ModelElementABS): (m ∈ s_3) ∧ name(m) = n)}
  IN
  (∀ (e: AssociationEndME):
     (e ∈ s_1) ⊃ ¬ (name(isModelElement(e)) ∈ (s_4 ∪ s_5)))

ClassifierWFR6PRED(c: ClassifierABS): boolean =
   (∀ (c_2: ClassifierABS, p_1: OperationME):
      (c_2 ∈ specificationsAUX(c)) ∧
       (isaFeature(isaBehavioralFeature(p_1)) ∈ allOperationsAsFeaturesAUX(c_2))
       ⊃
       (∃ (p_2: OperationME):
          (isaFeature(isaBehavioralFeature(p_2)) ∈ allOperationsAsFeaturesAUX(c)) ∧
           hasSameSignatureAUX(isaBehavioralFeature(p_1),
                               isaBehavioralFeature(p_2))))

ClassifierWFR7PRED(c: ClassifierABS): boolean =
   (∀ (g_1, g_2: GeneralizationME):
      (g_1 ∈ powertypeRangeASS(c)) ∧ (g_2 ∈ powertypeRangeASS(c)) ⊃
       discriminator(g_1) = discriminator(g_2))

ClassifierWFR8PRED(c: ClassifierABS): boolean =
   LET s_1: finite_set[NameCC]
          = {n: NameCC |
                 ∃ (a: AttributeME):
                   (a ∈ allAttributesAUX(c)) ∧
                    n = name(isaModelElement(isaFeature(isaStructuralFeature(a))))},
       s_2: finite_set[NameCC]
          = {n: NameCC |
                 ∃ (e: AssociationEndME):
                   (e ∈ allOppositeAssociationEndsAUX(c)) ∧
                    n = name(isaModelElement(e))}
     IN empty?((allDiscriminatorsAUX(c) ∩ (s_1 ∪ s_2)))

ClassifierWFRLEMMA: LEMMA
  (∀ (c: ClassifierABS):
     ClassifierWFR1PRED(c) ∧
      ClassifierWFR2PRED(c) ∧
       ClassifierWFR3PRED(c) ∧
        ClassifierWFR4PRED(c) ∧
         ClassifierWFR5PRED(c) ∧
          ClassifierWFR6PRED(c) ∧
           ClassifierWFR7PRED(c) ∧ ClassifierWFR8PRED(c))
```

```
ComponentWFR1PRED(c: ComponentME): boolean =
    (∀ (m: ModelElementABS):
        (m ∈ allContentsAUX(c)) ⊃
         isKindOf(isElement(m))(ComponentME))

ComponentWFR2PRED(c: ComponentME): boolean =
    (∀ (m: ModelElementABS):
        (m ∈ allResidentElementsAUX(c)) ⊃
         isKindOf(isElement(m))(DataTypeME) ∨
          isKindOf(isElement(m))(InterfaceME) ∨
           isKindOf(isElement(m))(ClassME) ∨
            isKindOf(isElement(m))(AssociationME) ∨
             isKindOf(isElement(m))(DependencyABS) ∨
              isKindOf(isElement(m))(ConstraintME) ∨
               isKindOf(isElement(m))(SignalME) ∨
                isKindOf(isElement(m))(DataValueME) ∨
                 isKindOf(isElement(m))(ObjectME))

ComponentWFRLEMMA: LEMMA
  (∀ (c: ComponentME): ComponentWFR1PRED(c) ∧ ComponentWFR2PRED(c))

ConstraintWFR1PRED(c: ConstraintME): boolean =
    ¬ (isModelElement(c) ∈ finseq2list(constrainedElementsASS(c)))

ConstraintWFRLEMMA: LEMMA (∀ (c: ConstraintME): ConstraintWFR1PRED(c))

DataTypeWFR1PRED(d: DataTypeME): boolean =
    (∀ (f: FeatureABS):
        (f ∈ allFeaturesAUX(isClassifier(d))) ⊃
         isKindOf(isElement(isModelElement(f)))(OperationME))
     ∧
     (∀ (p: OperationME):
         (isFeature(isBehavioralFeature(p)) ∈ allFeaturesAUX(isClassifier(d))) ⊃
          isQuery(isBehavioralFeature(p)))

DataTypeWFR2PRED(d: DataTypeME): boolean =
    empty?(allContentsAUX(isNameSpace(isClassifier(d))))

DataTypeWFRLEMMA: LEMMA
  (∀ (d: DataTypeME): DataTypeWFR1PRED(d) ∧ DataTypeWFR2PRED(d))

GeneralizableElementWFR1PRED(g: GeneralizableElementABS): boolean =
    (isRoot(g)) ⊃ empty?(generalizationsASS(g))
```

GeneralizableElementWFR2PRED (g : GeneralizableElementABS) : boolean =
(∀ (g_1 : GeneralizableElementABS) :
($g_1 \in$ parentsAUX(g)) $\supset$ ¬ (isLeaf (g_1)))

GeneralizableElementWFR3PRED (g : GeneralizableElementABS) : boolean =
¬ ($g \in$ allParentsAUX(g))

GeneralizableElementWFR4PRED (g : GeneralizableElementABS) : boolean =
(∀ (z : GeneralizationME) :
($z \in$ generalizationsASS(g)) $\supset$
(isaModelElement(parentASS(z)) $\in$ allContentsAUX(namespaceASS(isaModelElement(g)))))

GeneralizableElementWFRLEMMA : LEMMA
(∀ (g : GeneralizableElementABS) :
GeneralizableElementWFR1PRED (g) ∧
GeneralizableElementWFR2PRED (g) ∧
GeneralizableElementWFR3PRED (g) ∧ GeneralizableElementWFR4PRED (g))

GeneralizationWFR1PRED (g : GeneralizationME) : boolean =
(isKindOf (isaElement (isaModelElement (childASS (g)))) (ClassifierABS) ∧
isKindOf (isaElement (isaModelElement (parentASS (g)))) (ClassifierABS))
∨
(isKindOf (isaElement (isaModelElement (childASS (g)))) (ClassME) ∧
isKindOf (isaElement (isaModelElement (parentASS (g)))) (ClassME))
∨
(isKindOf (isaElement (isaModelElement (childASS (g)))) (InterfaceME) ∧
isKindOf (isaElement (isaModelElement (parentASS (g)))) (InterfaceME))
∨
(isKindOf (isaElement (isaModelElement (childASS (g)))) (DataTypeME) ∧
isKindOf (isaElement (isaModelElement (parentASS (g)))) (DataTypeME))
∨
(isKindOf (isaElement (isaModelElement (childASS (g)))) (NodeME) ∧
isKindOf (isaElement (isaModelElement (parentASS (g)))) (NodeME))
∨
(isKindOf (isaElement (isaModelElement (childASS (g)))) (ComponentME) ∧
isKindOf (isaElement (isaModelElement (parentASS (g)))) (ComponentME))
∨
(isKindOf (isaElement (isaModelElement (childASS (g)))) (AssociationME) ∧
isKindOf (isaElement (isaModelElement (parentASS (g)))) (AssociationME))
∨
(isKindOf (isaElement (isaModelElement (childASS (g)))) (AssociationClassME) ∧
isKindOf (isaElement (isaModelElement (parentASS (g))))
(AssociationClassME))

GeneralizationWFRLEMMA : LEMMA

```
  (∀ (g: GeneralizationME): GeneralizationWFR1PRED(g))

ImplementationClassWFR1PRED(c: ClassME): boolean =
    stereotypeAUX(c) = implementationClassSTE ⊃
     LET s: finite_set[ClassifierABS]
           = {c1: ClassifierABS |
                (∀ (i: InstanceME):
                   (i ∈ instancesASS(isClassifier(c))) ∧ (c1 ∈ classifiersASS(i)))}
       IN
       (∀ (c2: ClassifierABS):
          LET s: StereotypeME
                = stereotypeASS(isModelElement(isGeneralizableElement(c2)))
            IN
            (c2 ∈ s) ∧
             body(name(isModelElement(isGeneralizableElement(s))))
                = "implementationClass"
             ⊃ c2 = isClassifier(c))

ImplementationClassWFR2PRED(c: ClassME): boolean =
    stereotypeAUX(c) = implementationClassSTE ⊃
     (∀ (g: GeneralizableElementABS):
        (g ∈ parentsAUX(isGeneralizableElement(isClassifier(c)))) ⊃
         body(name(isModelElement(g))) = "implementationClass")

ImplementationClassWFRLEMMA: LEMMA
  (∀ (c: ClassME):
     stereotypeAUX(c) = implementationClassSTE ⊃
      ImplementationClassWFR1PRED(c) ∧ ImplementationClassWFR2PRED(c))

InterfaceWFR1PRED(i: InterfaceME): boolean =
    (∀ (f: FeatureABS):
       (f ∈ allFeaturesAUX(isClassifier(i))) ⊃
        isKindOf(isElement(isModelElement(f)))(OperationME) ∨
         isKindOf(isElement(isModelElement(f)))(ReceptionME))

InterfaceWFR2PRED(i: InterfaceME): boolean =
    empty?(allContentsAUX(isNameSpace(isClassifier(i))))

InterfaceWFR3PRED(i: InterfaceME): boolean =
    (∀ (f: FeatureABS):
       (f ∈ allFeaturesAUX(isClassifier(i))) ⊃
        visibility(f) = public)

InterfaceWFRLEMMA: LEMMA
  (∀ (i: InterfaceME):
```

```
    InterfaceWFR1PRED(i) ∧ InterfaceWFR2PRED(i) ∧ InterfaceWFR3PRED(i))

MethodWFR1PRED(m: MethodME): boolean =
    isQuery(isaBehavioralFeature(specificationASS(m))) ⊃
     isQuery(isaBehavioralFeature(m))

MethodWFR2PRED(m: MethodME): boolean =
    hasSameSignatureAUX(isaBehavioralFeature(m),
                        isaBehavioralFeature(specificationASS(m)))

MethodWFR3PRED(m: MethodME): boolean =
    visibility(isaFeature(isaBehavioralFeature(m))) =
     visibility(isaFeature(isaBehavioralFeature(specificationASS(m))))

MethodWFR4PRED(m: MethodME): boolean =
    LET s: finite_set[FeatureABS]
          = allOperationsAsFeaturesAUX(ownerASS(isaFeature(isaBehavioralFeature(m))))
      IN (isaFeature(isaBehavioralFeature(specificationASS(m))) ∈ s)

MethodWFR5PRED(m: MethodME): boolean =
    LET s1: finite_set[FeatureABS]
          = allOperationsAsFeaturesAUX(ownerASS(isaFeature(isaBehavioralFeature(m)))),
        s2: finite_set[FeatureABS]
          = {f: FeatureABS |
                 (f ∈ s1) ∧
                  (∃ (b: BehavioralFeatureABS):
                     isaFeature(b) = f ∧
                       hasSameSignatureAUX(b, isaBehavioralFeature(m)))},
        s3: finite_set[FeatureABS]
          = allOperationsAsFeaturesAUX(ownerASS(isaFeature
                                   (isaBehavioralFeature(specificationASS(m)))))
      IN (s3 ⊆ s2)

MethodWFRLEMMA: LEMMA
  (∀ (m: MethodME):
     MethodWFR1PRED(m) ∧
      MethodWFR2PRED(m) ∧
       MethodWFR3PRED(m) ∧ MethodWFR4PRED(m) ∧ MethodWFR5PRED(m))

NameSpaceWFR1PRED(ns: NameSpaceABS): boolean =
    (∀ (m1, m2: ModelElementABS):
       (m1 ∈ allContentsAUX(ns)) ∧
        (m2 ∈ allContentsAUX(ns)) ∧
         ¬ isKindOf(isaElement(m1))(AssociationME) ∧
          ¬ isKindOf(isaElement(m2))(AssociationME) ∧ name(m1) = name(m2)
```

```
    ⊃ m1 = m2)

NameSpaceWFR2PRED(ns: NameSpaceABS): boolean =
   LET s: finite_set[ModelElementABS]
          = {m: ModelElementABS |
               (m ∈ allContentsAUX(ns)) ∧ isKindOf(isElement(m))(AssociationME)}
     IN
     (∀ (a1, a2: AssociationME):
        (isModelElement(isRelationship(a1)) ∈ s) ∧
         (isModelElement(isRelationship(a2)) ∈ s) ∧
          name(isModelElement(isRelationship(a1))) =
           name(isModelElement(isRelationship(a2)))
           ∧
           length(connections(a1)) = length(connections(a2)) ∧
            (∀ (e1: AssociationEndME):
               (e1 ∈ finseq2list(connections(a1))) ⊃
                 (∃ (e2: AssociationEndME):
                    (e2 ∈ finseq2list(connections(a2))) ∧ typeASS(e1) = typeASS(e2)))
          ⊃ a1 = a2)

NameSpaceWFRLEMMA: LEMMA
  (∀ (n: NameSpaceABS): NameSpaceWFR1PRED(n) ∧ NameSpaceWFR2PRED(n))

StructuralFeatureWFR1PRED(f: StructuralFeatureABS): boolean =
   LET n: NameSpaceABS
       = namespaceASS(isModelElement(isGeneralizableElement(ownerASS(isFeature(f)))))
     IN
     (isModelElement(isGeneralizableElement(typeASS(f))) ∈ allContentsAUX(n))

StructuralFeatureWFR2PRED(f: StructuralFeatureABS): boolean =
   isKindOf(isElement(isModelElement(isGeneralizableElement(typeASS(f)))))(ClassME) ∨
    isKindOf(isElement(isModelElement(isGeneralizableElement(typeASS(f)))))
        (DataTypeME) ∨
     isKindOf(isElement(isModelElement(isGeneralizableElement(typeASS(f)))))
         (InterfaceME)

StructuralFeatureWFRLEMMA: LEMMA
  (∀ (f: StructuralFeatureABS):
     StructuralFeatureWFR1PRED(f) ∧ StructuralFeatureWFR2PRED(f))

TypeWFR1PRED(c: ClassME): boolean =
   stereotypeAUX(c) = typeSTE ⊃
    (∀ (f: FeatureABS):
       (f ∈ finseq2list(features(isClassifier(c)))) ⊃
        ¬ isKindOf(isElement(isModelElement(f)))(MethodME))
```

```
TypeWFR2PRED(c: ClassME): boolean =
    stereotypeAUX(c) = typeSTE ⊃
      (∀ (g: GeneralizableElementABS):
         (g ∈ parentsAUX(isaGeneralizableElement(isaClassifier(c)))) ⊃
          body(name(isaModelElement(g))) = "type")

TypeWFRLEMMA: LEMMA
   (∀ (c: ClassME):
      stereotypeAUX(c) = typeSTE ⊃ TypeWFR1PRED(c) ∧ TypeWFR2PRED(c))
END core_wfr
```

B.1.8 UML Package *Extension Mechanisms*

```
extensionMechanisms_abs : THEORY
 BEGIN

  IMPORTING backbone_abs

  TaggedValueCC : TYPE = [# tag : NameCC, value : StringCC #]

  StereotypeME : TYPE =
  [# isaGeneralizableElement : GeneralizableElementABS,
     icon : GeometryCC,
     baseClass : NameCC,
     requiredTags : finite_set[TaggedValueCC] #]

  stereotypeConstraintsASS : [StereotypeME → finite_set[ConstraintME]]

  extendedElementsASS : [StereotypeME → finite_set[ModelElementABS]]

  stereotypeASS : [ModelElementABS → StereotypeME]

  taggedValuesASS : [ModelElementABS → finite_set[TaggedValueCC]]
 END extensionMechanisms_abs
```

```
extensionMechanisms_wfr: THEORY
 BEGIN

  IMPORTING extensionMechanisms_abs, core_aux

  StereotypeWFR1PRED(s: StereotypeME): boolean =
      baseClass(s) ≠ name(isaModelElement(isaGeneralizableElement(s)))

  StereotypeWFR2PRED(s: StereotypeME): boolean =
      (∀ (g: GeneralizableElementABS):
         (g ∈ allParentsAUX(isaGeneralizableElement(s))) ∧
          (∃ (s2: StereotypeME):
             isaGeneralizableElement(s2) = g ⊃
              name(isaModelElement(isaGeneralizableElement(s))) ≠
               name(isaModelElement(isaGeneralizableElement(s2)))))

  StereotypeWFR4PRED(s: StereotypeME): boolean = body(baseClass(s)) ≠ ""

  StereotypeWFRLEMMA: LEMMA
    (∀ (s: StereotypeME):
       StereotypeWFR1PRED(s) ∧
        StereotypeWFR2PRED(s) ∧ StereotypeWFR4PRED(s))

  ModelElementWFR2PRED(me: ModelElementABS): boolean =
      (∀ (t1, t2: TaggedValueCC):
         (t1 ∈ taggedValuesASS(me)) ∧ (t2 ∈ taggedValuesASS(me)) ∧ tag(t1) = tag(t2) ⊃
          t1 = t2)

  ModelElementWFR3PRED(me: ModelElementABS): boolean =
      (∀ (t1: TaggedValueCC):
         (t1 ∈ requiredTags(stereotypeASS(me))) ∧ value(t1) = UNDEFINED ⊃
          (∃ (t2: TaggedValueCC):
             (t2 ∈ taggedValuesASS(me)) ∧ tag(t1) = tag(t2)))

  ModelElementWFRLEMMA: LEMMA
    (∀ (me: ModelElementABS):
       ModelElementWFR2PRED(me) ∧ ModelElementWFR3PRED(me))
 END extensionMechanisms_wfr
```

B.2 PVS Theories for UML Package *Behavioral Elements*

B.2.1 UML Subpackage *Common Behavior – Signals*

```
signals_abs: THEORY
 BEGIN

  IMPORTING backbone_abs

  SignalME: TYPE = [# isaClassifier: ClassifierABS #]

  ExceptionME: TYPE = [# isaSignal: SignalME #]

  ReceptionME: TYPE =
  [# isaBehavioralFeature: BehavioralFeatureABS,
     specification: StringCC,
     isRoot: boolean,
     isLeaf: boolean,
     isAbstract: boolean #]

  contextsASS: [SignalME → finite_set[BehavioralFeatureABS]]

  receptionsASS: [SignalME → finite_set[ReceptionME]]

  signalASS: [ReceptionME → SignalME]

  raisedSignalsASS: [BehavioralFeatureABS → finite_set[SignalME]]
 END signals_abs
```

B.2.2 UML Subpackage *Common Behavior – Actions*

```
actions_abs: THEORY
 BEGIN

  IMPORTING backbone_abs, signals_abs

  ArgumentME: TYPE = [# value: ExpressionCC #]

  ActionABS: TYPE =
  [# isaModelElement: ModelElementABS,
     recurrence: IterationExpressionCC,
     target: ObjectSetExpressionCC,
     isAsynchronous: BooleanCC,
     script: ActionExpressionCC,
     actualArguments: finite_sequence[ArgumentME] #]

  ActionSequenceME: TYPE =
  [# isaAction: ActionABS, actions: finite_sequence[ActionABS] #]

  CreateActionME: TYPE = [# isaAction: ActionABS #]

  CallActionME: TYPE = [# isaAction: ActionABS #]

  ReturnActionME: TYPE = [# isaAction: ActionABS #]

  SendActionME: TYPE = [# isaAction: ActionABS #]

  TerminateActionME: TYPE = [# isaAction: ActionABS #]

  UninterpretedActionME: TYPE = [# isaAction: ActionABS #]

  DestroyActionME: TYPE = [# isaAction: ActionABS #]

  instantiationASS: [CreateActionME -> ClassifierABS]

  operationASS: [CallActionME -> OperationME]

  signalASS: [SendActionME -> SignalME]
 END actions_abs
```

B.2.3 UML Subpackage *Common Behavior – Instances And Links*

```
instancesAndLinks_abs: THEORY
 BEGIN

  IMPORTING backbone_abs, relationships_abs, signals_abs, actions_abs

  AttributeLinkME: TYPE =
  [# isaModelElement: ModelElementABS, attributeASS: AttributeME #]

  InstanceME: TYPE =
  [# isaModelElement: ModelElementABS,
     slots: finite_set[AttributeLinkME] #]

  StimulusME: TYPE =
  [# isaModelElement: ModelElementABS,
     arguments: finite_sequence[InstanceME] #]

  LinkEndME: TYPE = [# isaModelElement: ModelElementABS #]

  LinkME: TYPE =
  [# isaModelElement: ModelElementABS,
     connections: finite_sequence[LinkEndME] #]

  DataValueME: TYPE = [# isaInstance: InstanceME #]

  ComponentInstanceME: TYPE =
  [# isaInstance: InstanceME, residents: finite_set[InstanceME] #]

  NodeInstanceME: TYPE =
  [# isaInstance: InstanceME, residents: finite_set[ComponentInstanceME] #]

  ObjectME: TYPE = [# isaInstance: InstanceME #]

  LinkObjectME: TYPE = [# isaLink: LinkME, isaObject: ObjectME #]

  valueASS: [AttributeLinkME → InstanceME]

  classifiersASS: [InstanceME → finite_set[ClassifierABS]]

  linkEndsASS: [InstanceME → finite_set[LinkEndME]]

  receiverASS: [StimulusME → InstanceME]

  senderASS: [StimulusME → InstanceME]
```

dispatchActionASS : [StimulusME → ActionABS]

communicationLinkASS : [StimulusME → LinkME]

instanceASS : [LinkEndME → InstanceME]

associationEndASS : [LinkEndME → AssociationEndME]

linkASS : [LinkEndME → LinkME]

associationASS : [LinkME → AssociationME]

instancesASS : [ClassifierABS → finite_set[InstanceME]]

linksASS : [AssociationME → finite_set[LinkME]]
END instancesAndLinks_abs

B.2.4 UML Package *Common Behavior*

```
commonBehavior_aux: THEORY
 BEGIN

  IMPORTING backbone_abs, signals_abs, actions_abs, instancesAndLinks_abs, core_aux

  allLinksAUX(i: InstanceME): finite_set[LinkME] =
      {l: LinkME |
           ∃ (e: LinkEndME): (e ∈ linkEndsASS(i)) ∧ l = linkASS(e)}

  allOppositeLinkEndsAUX(i: InstanceME): finite_set[LinkEndME] =
      {e: LinkEndME |
           ∃ (l: LinkME):
             (l ∈ allLinksAUX(i)) ∧
              (e ∈ finseq2list(connections(l))) ∧ instanceASS(e) ≠ i}

  selectedLinkEndsAUX(i: InstanceME, a: AssociationEndME): finite_set[LinkEndME] =
      {e: LinkEndME |
           (e ∈ allOppositeLinkEndsAUX(i)) ∧ associationEndASS(e) = a}

  selectedAttributeLinksAUX(i: InstanceME, a: AttributeME): finite_set[AttributeLinkME] =
      {l: AttributeLinkME | (l ∈ slots(i)) ∧ attributeASS(l) = a}
 END commonBehavior_aux
```

```
commonBehavior_wfr: THEORY
 BEGIN

  IMPORTING commonBehavior_aux

  AttributeLinkWFR1PRED(a: AttributeLinkME): boolean =
      LET s1: finite_set[ClassifierABS] = classifiersASS(valueASS(a)),
          s2: finite_set[ClassifierABS]
            = {c2: ClassifierABS |
                   ∃ (c1: ClassifierABS): (c1 ∈ s1) ∧ (c2 ∈ allParentsAUX(c1))}
        IN (typeASS(isaStructuralFeature(attributeASS(a))) ∈ (s1 ∪ s2))

  AttributeLinkWFRLEMMA: LEMMA
    (∀ (a: AttributeLinkME): AttributeLinkWFR1PRED(a))

  CallActionWFR1PRED(a: CallActionME): boolean =
      length(actualArguments(isaAction(a))) =
       length(parameters(isaBehavioralFeature(operationASS(a))))

  CallActionWFRLEMMA: LEMMA (∀ (a: CallActionME): CallActionWFR1PRED(a))

  ComponentInstanceWFR1PRED(i: ComponentInstanceME): boolean =
      card(classifiersASS(isaInstance(i))) = 1 ∧
       (∀ (c: ClassifierABS):
          (c ∈ classifiersASS(isaInstance(i))) ⊃
           isKindOf(isaElement(isaModelElement(isaGeneralizableElement(c))))
                (ComponentME))

  ComponentInstanceWFRLEMMA: LEMMA
    (∀ (i: ComponentInstanceME): ComponentInstanceWFR1PRED(i))

  CreateActionWFR1PRED(a: CreateActionME): boolean =
      body(isaExpression(target(isaAction(a)))) = extract1(empty_seq)

  CreateActionWFRLEMMA: LEMMA
    (∀ (a: CreateActionME): CreateActionWFR1PRED(a))

  DestroyActionWFR1PRED(a: DestroyActionME): boolean =
      length(actualArguments(isaAction(a))) = 0

  DestroyActionWFRLEMMA: LEMMA
    (∀ (a: DestroyActionME): DestroyActionWFR1PRED(a))

  DataValueWFR1PRED(v: DataValueME): boolean =
      card(classifiersASS(isaInstance(v))) = 1 ∧
```

```
(∀ (c: ClassifierABS):
    (c ∈ classifiersASS(isaInstance(v))) ⊃
     isKindOf(isaElement(isaModelElement(isaGeneralizableElement(c))))
          (DataTypeME))

DataValueWFR2PRED(v: DataValueME): boolean =
    empty?(slots(isaInstance(v)))

DataValueWFRLEMMA: LEMMA
  (∀ (v: DataValueME): DataValueWFR1PRED(v) ∧ DataValueWFR2PRED(v))

InstanceWFR1PRED(i: InstanceME): boolean =
    (∀ (a: AttributeLinkME):
        (a ∈ slots(i)) ⊃
         (∃ (c: ClassifierABS):
             (c ∈ classifiersASS(i)) ∧
              (attributeASS(a) ∈ allAttributesAUX(c))))

InstanceWFR2PRED(i: InstanceME): boolean =
    (∀ (l: LinkME):
        (l ∈ allLinksAUX(i)) ⊃
         (∃ (c: ClassifierABS):
             (c ∈ classifiersASS(i)) ∧
              (associationASS(l) ∈ allAssociationsAUX(c))))

InstanceWFR3PRED(i: InstanceME): boolean =
    (∀ (c1, c2: ClassifierABS, p1, p2: OperationME):
        (c1 ∈ classifiersASS(i)) ∧
         (c2 ∈ classifiersASS(i)) ∧
          (p1 ∈ allOperationsAUX(c1)) ∧
           (p2 ∈ allOperationsAUX(c2)) ∧
            hasSameSignatureAUX(isaBehavioralFeature(p1), isaBehavioralFeature(p2))
         ⊃ p1 = p2)

InstanceWFR4PRED(i: InstanceME): boolean =
    (∀ (a: AttributeLinkME):
        (a ∈ slots(i)) ⊃
         (¬ (∃ (e: LinkEndME):
                (e ∈ allOppositeLinkEndsAUX(i)) ∧
                 name(isaModelElement(e)) = name(isaModelElement(a)))))
     ∧
     (∀ (e: LinkEndME):
         (e ∈ allOppositeLinkEndsAUX(i)) ⊃
          (¬ (∃ (a: AttributeLinkME):
                 (a ∈ slots(i)) ∧
```

```
                name(isaModelElement(e)) = name(isaModelElement(a)))))

InstanceWFR5PRED(i: InstanceME): boolean =
    LET s: finite_set[AssociationEndME]
          = {e: AssociationEndME |
                (∃ (c: ClassifierABS):
                   (c ∈ classifiersASS(i)) ∧ (e ∈ allOppositeAssociationEndsAUX(c)))}
      IN
      (∀ (e₂: AssociationEndME):
         (e₂ ∈ s) ⊃
          (∃ (r: MultiplicityRangeCC):
             (r ∈ ranges(multiplicity(e₂))) ∧
              card(selectedLinkEndsAUX(i, e₂)) ≥ lower(r) ∧
               (upper(r) = UNLIMITED ∨
                 (upper(r) ≠ UNLIMITED ∧
                   card(selectedLinkEndsAUX(i, e₂)) ≤ upper(r)))))

InstanceWFR6PRED(i: InstanceME): boolean =
    LET s: finite_set[AttributeME]
          = {a: AttributeME |
                (∃ (c: ClassifierABS):
                   (c ∈ classifiersASS(i)) ∧ (a ∈ allAttributesAUX(c)))}
      IN
      (∀ (a₂: AttributeME):
         (a₂ ∈ s) ⊃
          (∃ (r: MultiplicityRangeCC):
             (r ∈ ranges(multiplicity(isaStructuralFeature(a₂)))) ∧
              card(selectedAttributeLinksAUX(i, a₂)) ≥ lower(r) ∧
               (upper(r) = UNLIMITED ∨
                 (upper(r) ≠ UNLIMITED ∧
                   card(selectedAttributeLinksAUX(i, a₂)) ≤ upper(r)))))

InstanceWFRLEMMA: LEMMA
  (∀ (i: InstanceME):
      InstanceWFR1PRED(i) ∧
       InstanceWFR2PRED(i) ∧
        InstanceWFR3PRED(i) ∧
         InstanceWFR4PRED(i) ∧ InstanceWFR5PRED(i) ∧ InstanceWFR6PRED(i))

LinkWFR1PRED(l: LinkME): boolean =
    (∀ (i: below(length(connections(l)))):
        associationEndASS(nth(finseq2list(connections(l)), i)) =
         nth(finseq2list(connections(associationASS(l))), i))

LinkWFR2PRED(l: LinkME): boolean =
```

```
    LET s: finite_set[LinkME] = linksASS(associationASS(l)) IN
      (∀ (l2: LinkME):
         (l2 ∈ s) ∧
          (∀ (i: below(length(connections(l)))):
              instanceASS(nth(finseq2list(connections(l)), i)) =
                instanceASS(nth(finseq2list(connections(l2)), i)))
          ⊃ l = l2)

LinkWFRLEMMA: LEMMA (∀ (l: LinkME): LinkWFR1PRED(l) ∧ LinkWFR2PRED(l))

LinkEndWFR1PRED(e: LinkEndME): boolean =
    LET s1: finite_set[ClassifierABS] = classifiersASS(instanceASS(e)),
        s2: finite_set[ClassifierABS]
            = {c2: ClassifierABS |
                    ∃ (c1: ClassifierABS): (c1 ∈ s1) ∧ (c2 ∈ allParentsAUX(c1))}
      IN (typeASS(associationEndASS(e)) ∈ (s1 ∪ s2))

LinkEndWFRLEMMA: LEMMA (∀ (e: LinkEndME): LinkEndWFR1PRED(e))

LinkObjectWFR1PRED(l: LinkObjectME): boolean =
    (∃ (a: AssociationClassME):
       isAssociation(a) = associationASS(isaLink(l)) ∧
        (isaClassifier(isaClass(a)) ∈ classifiersASS(isaInstance(isaObject(l)))))

LinkObjectWFR2PRED(l: LinkObjectME): boolean =
    isKindOf(isaElement(isaModelElement(isaRelationship(associationASS(isaLink(l))))))
         (AssociationClassME)

LinkObjectWFRLEMMA: LEMMA
  (∀ (l: LinkObjectME): LinkObjectWFR1PRED(l) ∧ LinkObjectWFR2PRED(l))

NodeInstanceWFR1PRED(i: NodeInstanceME): boolean =
    (∀ (c: ClassifierABS):
       (c ∈ classifiersASS(isaInstance(i))) ⊃
        isKindOf(isaElement(isaModelElement(isaGeneralizableElement(c))))(NodeME) ∧
         card(classifiersASS(isaInstance(i))) = 1)

NodeInstanceWFR2PRED(i: NodeInstanceME): boolean =
    (∀ (i2: ComponentInstanceME):
       (i2 ∈ residents(i)) ⊃
        LET s1: finite_set[ComponentME]
                = {c: ComponentME | (isaClassifier(c) ∈ classifiersASS(isaInstance(i2)))},
            s2: finite_set[NodeME]
                = {n: NodeME | (isaClassifier(n) ∈ classifiersASS(isaInstance(i)))},
            s3: finite_set[ComponentME]
```

= $\{c\colon \text{ComponentME} \mid (\exists\ (n\colon \text{NodeME}) : (n \in s_2) \wedge (c \in \text{residents}(n)))\}$
IN $(s_1 \subseteq s_3)$)

NodeInstanceWFRLEMMA: LEMMA
 (∀ (i: NodeInstanceME):
  NodeInstanceWFR1PRED(i) ∧ NodeInstanceWFR2PRED(i))

ObjectWFR1PRED(b: ObjectME): boolean =
 (∀ (c: ClassifierABS):
  $(c \in \text{classifiersASS}(\text{isaInstance}(b))) \supset$
  isKindOf(isaElement(isaModelElement(isaGeneralizableElement(c))))
   (ClassME))

ObjectWFRLEMMA: LEMMA (∀ (b: ObjectME): ObjectWFR1PRED(b))

ReceptionWFR1PRED(r: ReceptionME): boolean =
 ¬ isQuery(isaBehavioralFeature(r))

ReceptionWFRLEMMA: LEMMA (∀ (r: ReceptionME): ReceptionWFR1PRED(r))

SendActionWFR1PRED(a: SendActionME): boolean =
 length(actualArguments(isaAction(a))) =
  card(allAttributesAUX(isaClassifier(signalASS(a))))

SendActionWFR2PRED(a: SendActionME): boolean =
 isAsynchronous(isaAction(a))

SendActionWFRLEMMA: LEMMA
 (∀ (a: SendActionME): SendActionWFR1PRED(a) ∧ SendActionWFR2PRED(a))

StimulusWFR1PRED(s: StimulusME): boolean =
 length(actualArguments(dispatchActionASS(s))) = length(arguments(s))

StimulusWFR2PRED(s: StimulusME): boolean =
 isKindOf(isaElement(isaModelElement(dispatchActionASS(s))))(SendActionME) ∨
 isKindOf(isaElement(isaModelElement(dispatchActionASS(s))))(CallActionME) ∨
 isKindOf(isaElement(isaModelElement(dispatchActionASS(s))))(CreateActionME) ∨
 isKindOf(isaElement(isaModelElement(dispatchActionASS(s))))
  (DestroyActionME)

StimulusWFRLEMMA: LEMMA
 (∀ (s: StimulusME): StimulusWFR1PRED(s) ∧ StimulusWFR2PRED(s))

TerminateActionWFR1PRED(a: TerminateActionME): boolean =
 length(actualArguments(isaAction(a))) = 0

```
 TerminateActionWFR2PRED(a: TerminateActionME) : boolean =
      body(isaExpression(target(isaAction(a)))) = extract1(empty_seq)

 TerminateActionWFRLEMMA: LEMMA
    (∀ (a: TerminateActionME) :
       TerminateActionWFR1PRED(a) ∧ TerminateActionWFR2PRED(a))
END commonBehavior_wfr
```

B.2.5 UML Package *Collaborations*

```
collaborations_abs: THEORY
 BEGIN

  IMPORTING backbone_abs, relationships_abs, actions_abs

  ClassifierRoleME: TYPE =
  [# isaClassifier: ClassifierABS,
     multiplicity: MultiplicityCC,
     availableFeatures: finite_set[FeatureABS],
     availableContents: finite_set[ModelElementABS] #]

  AssociationEndRoleME: TYPE =
  [# isaAssociationEnd: AssociationEndME,
     collaborationMultiplicity: MultiplicityCC,
     availableQualifiers: finite_set[AttributeME] #]

  AssociationRoleME: TYPE =
  [# isaAssociation: AssociationME,
     multiplicity: MultiplicityCC,
     connections: finite_sequence[AssociationEndRoleME] #]

  MessageME: TYPE = [# isaModelElement: ModelElementABS #]

  InteractionME: TYPE =
  [# isaModelElement: ModelElementABS, messages: finite_set[MessageME] #]

  CollaborationME: TYPE =
  [# isaNameSpace: NameSpaceABS,
     isaGeneralizableElement: GeneralizableElementABS,
     interactions: finite_set[InteractionME] #]

  baseASS: [ClassifierRoleME -> ClassifierABS]

  typeASS: [AssociationEndRoleME -> ClassifierRoleME]

  baseASS: [AssociationEndRoleME -> AssociationEndME]

  baseASS: [AssociationRoleME -> AssociationME]

  communicationConnectionASS: [MessageME -> AssociationRoleME]

  actionASS: [MessageME -> ActionABS]
```

receiverASS : [MessageME → ClassifierRoleME]

senderASS : [MessageME → ClassifierRoleME]

predecessorsASS : [MessageME → finite_set[MessageME]]

activatorASS : [MessageME → MessageME]

interactionASS : [MessageME → InteractionME]

contextASS : [InteractionME → CollaborationME]

constrainingElementsASS : [CollaborationME → finite_set[ModelElementABS]]

representedOperationASS : [CollaborationME → OperationME]

representedClassifierASS : [CollaborationME → ClassifierABS]
END collaborations_abs

```
collaborations_aux: THEORY
 BEGIN

  IMPORTING backbone_abs, core_aux, collaborations_abs

  inheritanceHierarchyDepthAUX: [CollaborationME → nat]

  inheritanceHierarchyDepthAUX: [ClassifierRoleME → nat]

  messageHierarchyDepthAUX: [MessageME → nat]

  allAvailableFeaturesAUX(r: ClassifierRoleME): RECURSIVE finite_set[FeatureABS] =
    LET s: finite_set[FeatureABS]
           = {f: FeatureABS |
                  (∃ (c: ClassifierABS, r2: ClassifierRoleME):
                     (c ∈ parentsAUX(isaClassifier(r))) ∧
                      c = isaClassifier(r2) ∧ (f ∈ allAvailableFeaturesAUX(r2)))}
      IN (availableFeatures(r) ∪ s)
     MEASURE inheritanceHierarchyDepthAUX(r)

  allAvailableContentsAUX(r: ClassifierRoleME): RECURSIVE finite_set[ModelElementABS] =
    LET s: finite_set[ModelElementABS]
           = {m: ModelElementABS |
                  (∃ (c: ClassifierABS, r2: ClassifierRoleME):
                     (c ∈ parentsAUX(isaClassifier(r))) ∧
                      c = isaClassifier(r2) ∧ (m ∈ allAvailableContentsAUX(r2)))}
      IN (availableContents(r) ∪ s)
     MEASURE inheritanceHierarchyDepthAUX(r)

  allContentsAUX(c: CollaborationME): RECURSIVE finite_set[ModelElementABS] =
    LET s1: finite_set[ModelElementABS] = contentsAUX(isaNameSpace(c)),
        s2: finite_set[ModelElementABS]
           = {m: ModelElementABS |
                  (∃ (g: GeneralizableElementABS, c2: CollaborationME):
                     (g ∈ parentsAUX(isaGeneralizableElement(c))) ∧
                      g = isaGeneralizableElement(c2) ∧ (m ∈ allContentsAUX(c2)))},
        s3: finite_set[NameCC]
           = {n: NameCC | (∃ (m: ModelElementABS): (m ∈ s1) ∧ name(m) = n)},
        s4: finite_set[ModelElementABS]
           = {m: ModelElementABS | (m ∈ s2) ∧ (name(m) ∈ s3)}
      IN (s1 ∪ (s2 \ s4))
     MEASURE inheritanceHierarchyDepthAUX(c)

  allPredecessorsAUX(m: MessageME): RECURSIVE finite_set[MessageME] =
    LET s1: finite_set[MessageME] = predecessorsASS(m),
```

s_2 : finite_set [MessageME]
= $\{m_2$: MessageME |
$\exists$ (m_1 : MessageME) : $(m_1 \in s_1)$ $\wedge$ $(m_2 \in$ allPredecessorsAUX$(m_1))\}$
IN $(s_1 \cup s_2)$
MEASURE messageHierarchyDepthAUX (m)
END collaborations_aux

```
collaborations_wfr: THEORY
 BEGIN

  IMPORTING collaborations_aux

  AssociationEndRoleWFR1PRED(r: AssociationEndRoleME): boolean =
       baseASS(typeASS(r)) = typeASS(baseASS(r)) ∨
        (baseASS(typeASS(r)) ∈ allParentsAUX(typeASS(baseASS(r))))

  AssociationEndRoleWFR2PRED(r: AssociationEndRoleME): boolean =
       isKindOf(isaElement(isaModelElement(isaGeneralizableElement(isaClassifier(typeASS(r))))))
            (ClassifierRoleME)

  AssociationEndRoleWFR3PRED(r: AssociationEndRoleME): boolean =
       (∀ (a: AttributeME):
           (a ∈ availableQualifiers(r)) ⊃
            (a ∈ finseq2list(qualifiers(baseASS(r)))))

  AssociationEndRoleWFR4PRED(r: AssociationEndRoleME): boolean =
       isNavigable(isaAssociationEnd(r)) ⊃ isNavigable(baseASS(r))

  AssociationEndRoleWFRLEMMA: LEMMA
     (∀ (r: AssociationEndRoleME):
         AssociationEndRoleWFR1PRED(r) ∧
          AssociationEndRoleWFR2PRED(r) ∧
           AssociationEndRoleWFR3PRED(r) ∧ AssociationEndRoleWFR4PRED(r))

  AssociationRoleWFR1PRED(a: AssociationRoleME): boolean =
       (∀ (i: below(length(connections(a)))):
           baseASS(nth(finseq2list(connections(a)), i)) =
            nth(finseq2list(connections(baseASS(a))), i))

  AssociationRoleWFR2PRED(a: AssociationRoleME): boolean =
       (∀ (r: AssociationEndRoleME):
           (r ∈ finseq2list(connections(a))) ⊃
            isKindOf(isaElement(isaModelElement(isaAssociationEnd(r))))
                 (AssociationEndRoleME))

  AssociationRoleWFRLEMMA: LEMMA
     (∀ (a: AssociationRoleME):
         AssociationRoleWFR1PRED(a) ∧ AssociationRoleWFR2PRED(a))

  ClassifierRoleWFR1PRED(c: ClassifierRoleME): boolean =
       (∀ (r: AssociationRoleME):
           (isaAssociation(r) ∈ allAssociationsAUX(isaClassifier(c))) ⊃
```

```
        (∃ (a: AssociationME):
            (a ∈ allAssociationsAUX(baseASS(c))) ∧ baseASS(r) = a))

ClassifierRoleWFR2PRED(c: ClassifierRoleME): boolean =
    (allAvailableFeaturesAUX(c) ⊆ allFeaturesAUX(baseASS(c))) ∧
     (allAvailableContentsAUX(c) ⊆ allContentsAUX(baseASS(c)))

ClassifierRoleWFR3PRED(c: ClassifierRoleME): boolean =
    empty?(allFeaturesAUX(isaClassifier(c)))

ClassifierRoleWFRLEMMA: LEMMA
  (∀ (c: ClassifierRoleME):
     ClassifierRoleWFR1PRED(c) ∧
      ClassifierRoleWFR2PRED(c) ∧ ClassifierRoleWFR3PRED(c))

CollaborationWFR1PRED(l: CollaborationME): boolean =
    (∀ (c: ClassifierRoleME):
        (isaModelElement(isaGeneralizableElement(isaClassifier(c))) ∈ allContentsAUX(l)) ⊃
         (isaModelElement(isaGeneralizableElement(baseASS(c))) ∈
                  allContentsAUX(namespaceASS(isaModelElement(isaNameSpace(l))))))
     ∧
     (∀ (a: AssociationRoleME):
         (isaModelElement(isaRelationship(isaAssociation(a))) ∈ allContentsAUX(l)) ⊃
          (isaModelElement(isaRelationship(baseASS(a))) ∈
                  allContentsAUX(namespaceASS(isaModelElement(isaNameSpace(l))))))

CollaborationWFR2PRED(l: CollaborationME): boolean =
    (constrainingElementsASS(l) ⊆ allContentsAUX(namespaceASS(isaModelElement(isaNameSpace(l)))))

CollaborationWFR3PRED(l: CollaborationME): boolean =
    (∀ (c1: ClassifierRoleME):
        (isaModelElement(isaGeneralizableElement(isaClassifier(c1))) ∈ allContentsAUX(l)) ∧
         body(name(isaModelElement(isaGeneralizableElement(isaClassifier(c1))))) = ""
         ⊃
         (∀ (c2: ClassifierRoleME):
             (isaModelElement(isaGeneralizableElement(isaClassifier(c2))) ∈ allContentsAUX(l))
              ⊃ baseASS(c1) = baseASS(c2) ⊃ c1 = c2))
     ∧
     (∀ (a1: AssociationRoleME):
         (isaModelElement(isaRelationship(isaAssociation(a1))) ∈ allContentsAUX(l)) ∧
          body(name(isaModelElement(isaRelationship(isaAssociation(a1))))) = ""
          ⊃
          (∀ (a2: AssociationRoleME):
              (isaModelElement(isaRelationship(isaAssociation(a2))) ∈ allContentsAUX(l)) ⊃
               baseASS(a1) = baseASS(a2) ⊃ a1 = a2))
```

```
CollaborationWFR4PRED(l: CollaborationME): boolean =
    (∀ (m: ModelElementABS):
        (m ∈ ownedElements(isaNameSpace(l))) ⊃
         isKindOf(isaElement(m))(ClassifierRoleME) ∨
          isKindOf(isaElement(m))(AssociationRoleME) ∨
           isKindOf(isaElement(m))(GeneralizationME) ∨
            isKindOf(isaElement(m))(ConstraintME))

CollaborationWFR5PRED(l: CollaborationME): boolean =
    LET s1: finite_set[ModelElementABS] = contentsAUX(isaNameSpace(l)),
        s2: finite_set[ModelElementABS]
            = {m: ModelElementABS |
                    (∃ (g: GeneralizableElementABS, c2: CollaborationME):
                        (g ∈ parentsAUX(isaGeneralizableElement(l))) ∧
                         g = isaGeneralizableElement(c2) ∧ (m ∈ allContentsAUX(c2)))}
      IN
      (∀ (g1, g2: GeneralizableElementABS):
          (isaModelElement(g1) ∈ s1) ∧
           (isaModelElement(g2) ∈ s2) ∧
              name(isaModelElement(g1)) = name(isaModelElement(g2))
           ⊃ (g2 ∈ allParentsAUX(g1)))

CollaborationWFRLEMMA: LEMMA
   (∀ (l: CollaborationME):
       CollaborationWFR1PRED(l) ∧
        CollaborationWFR2PRED(l) ∧
         CollaborationWFR3PRED(l) ∧
          CollaborationWFR4PRED(l) ∧ CollaborationWFR5PRED(l))

InteractionWFR1PRED(i: InteractionME): boolean =
    (∀ (m: MessageME):
        (m ∈ messages(i)) ∧
           isKindOf(isaElement(isaModelElement(actionASS(m))))(SendActionME)
         ⊃
         (∃ (a: SendActionME):
            isaAction(a) = actionASS(m) ∧
             (isaModelElement(isaGeneralizableElement(isaClassifier(signalASS(a)))) ∈
               allContentsAUX(namespaceASS(isaModelElement(isaNameSpace(contextASS(i)))))))

InteractionWFRLEMMA: LEMMA (∀ (i: InteractionME): InteractionWFR1PRED(i))

MessageWFR1PRED(m: MessageME): boolean =
    (isaModelElement(isaGeneralizableElement(isaClassifier(senderASS(m)))) ∈
                    ownedElements(isaNameSpace(contextASS(interactionASS(m)))))
```

```
   ∧
   (isaModelElement(isaGeneralizableElement(isaClassifier(receiverASS(m)))) ∈
                  ownedElements(isaNameSpace(contextASS(interactionASS(m)))))

MessageWFR2PRED(m: MessageME): boolean =
   (∀ (m2: MessageME):
      (m2 ∈ predecessorsASS(m)) ⊃ interactionASS(m2) = interactionASS(m))
   ∧ interactionASS(activatorASS(m)) = interactionASS(m)

MessageWFR3PRED(m: MessageME): boolean =
   (∀ (m2: MessageME):
      (m2 ∈ allPredecessorsAUX(m)) ⊃
       activatorASS(m) = activatorASS(m2))

MessageWFR4PRED(m: MessageME): boolean =
   ¬ (m ∈ allPredecessorsAUX(m))

MessageWFR5PRED(m: MessageME): boolean =
   (isaModelElement(isaRelationship(isaAssociation(communicationConnectionASS(m)))) ∈
                  ownedElements(isaNameSpace(contextASS(interactionASS(m)))))

MessageWFR6PRED(m: MessageME): boolean =
   (∃ (r: AssociationEndRoleME):
      (r ∈ finseq2list(connections(communicationConnectionASS(m)))) ∧
       typeASS(r) = senderASS(m))
   ∧
   (∃ (r: AssociationEndRoleME):
      (r ∈ finseq2list(connections(communicationConnectionASS(m)))) ∧
       typeASS(r) = receiverASS(m))

MessageWFRLEMMA: LEMMA
  (∀ (m: MessageME):
     MessageWFR1PRED(m) ∧
      MessageWFR2PRED(m) ∧
       MessageWFR3PRED(m) ∧
        MessageWFR4PRED(m) ∧ MessageWFR5PRED(m) ∧ MessageWFR6PRED(m))
END collaborations_wfr
```

B.2.6 UML Package *State Machines*

```
stateMachines_abs: THEORY
 BEGIN

  IMPORTING backbone_abs, actions_abs

  StateVertexABS: TYPE = [# isaModelElement: ModelElementABS #]

  EventABS: TYPE =
  [# isaModelElement: ModelElementABS,
     parameters: finite_sequence[ParameterME] #]

  GuardME: TYPE =
  [# isaModelElement: ModelElementABS, expression: BooleanExpressionCC #]

  TransitionME: TYPE =
  [# isaModelElement: ModelElementABS,
     guard: finite_set[GuardME],
     effect: finite_set[ActionABS],
     trigger: finite_set[EventABS] #]

  StateME: TYPE =
  [# isaStateVertex: StateVertexABS,
     internals: finite_set[TransitionME],
     entry: finite_set[ActionABS],
     exit: finite_set[ActionABS],
     doActivity: finite_set[ActionABS] #]

  PseudoStateME: TYPE =
  [# isaStateVertex: StateVertexABS, kind: PseudostateKindCC #]

  SynchStateME: TYPE =
  [# isaStateVertex: StateVertexABS, bound: UnlimitedIntegerCC #]

  StubStateME: TYPE =
  [# isaStateVertex: StateVertexABS, referenceState: NameCC #]

  StateMachineME: TYPE =
  [# isaModelElement: ModelElementABS,
     transitions: finite_set[TransitionME],
     top: StateME #]

  CompositeStateME: TYPE =
  [# isaState: StateME,
```

```
    isConcurrent: BooleanCC,
    subvertices: finite_set[StateVertexABS] #]

SimpleStateME: TYPE = [# isaState: StateME #]

FinalStateME: TYPE = [# isaState: StateME #]

SubmachineStateME: TYPE = [# isaCompositeState: CompositeStateME #]

SignalEventME: TYPE = [# isaEvent: EventABS #]

CallEventME: TYPE = [# isaEvent: EventABS #]

TimeEventME: TYPE = [# isaEvent: EventABS, whenExpr: TimeExpressionCC #]

ChangeEventME: TYPE =
[# isaEvent: EventABS, changeExpression: BooleanExpressionCC #]

outgoingsASS: [StateVertexABS → finite_set[TransitionME]]

incomingsASS: [StateVertexABS → finite_set[TransitionME]]

containerASS: [StateVertexABS → finite_set[CompositeStateME]]

sourceASS: [TransitionME → StateVertexABS]

targetASS: [TransitionME → StateVertexABS]

stateMachineASS: [TransitionME → finite_set[StateMachineME]]

deferrableEventsASS: [StateME → finite_set[EventABS]]

contextASS: [StateMachineME → finite_set[ModelElementABS]]

submachineASS: [SubmachineStateME → StateMachineME]

signalASS: [SignalEventME → SignalME]

operationASS: [CallEventME → OperationME]

behaviorsASS: [ModelElementABS → finite_set[StateMachineME]]

occurrencesASS: [SignalME → finite_set[SignalEventME]]

occurrencesASS: [OperationME → finite_set[CallEventME]]
```

```
CallEventSTES: TYPE = {createSTE, destroySTE}

stereotypeAUX: [CallEventME -> CallEventSTES]
END stateMachines_abs
```

```
stateMachines_aux : THEORY
 BEGIN

  IMPORTING stateMachines_abs

  StateWFRPREDAUX(s: StateME) : boolean =
      card(entry(s)) ≤ 1 ∧ card(exit(s)) ≤ 1 ∧ card(doActivity(s)) ≤ 1

  StateWFRLEMMAAUX: LEMMA (∀ (s: StateME) : StateWFRPREDAUX(s))

  StateMachineWFRPREDAUX(m: StateMachineME) : boolean =
      card(contextASS(m)) ≤ 1

  StateMachineWFRLEMMAAUX: LEMMA
    (∀ (m: StateMachineME) : StateMachineWFRPREDAUX(m))

  StateVertexWFRPREDAUX(v: StateVertexABS) : boolean =
      card(containerASS(v)) ≤ 1

  StateVertexWFRLEMMAAUX: LEMMA
    (∀ (v: StateVertexABS) : StateVertexWFRPREDAUX(v))

  TransitionWFRPREDAUX(t: TransitionME) : boolean =
      card(trigger(t)) ≤ 1 ∧
       card(guard(t)) ≤ 1 ∧
        card(effect(t)) ≤ 1 ∧ card(stateMachineASS(t)) ≤ 1

  TransitionWFRLEMMAAUX: LEMMA
    (∀ (s: TransitionME) : TransitionWFRPREDAUX(s))
 END stateMachines_aux
```

```
stateMachines_wfr : THEORY
 BEGIN

  IMPORTING stateMachines_abs

  CompositeStateWFR1PRED(c: CompositeStateME) : boolean =
      LET s: finite_set[PseudoStateME]
            = {p: PseudoStateME | (isaStateVertex(p) ∈ subvertices(c)) ∧ kind(p) = initial}
        IN card(s) ≤ 1

  CompositeStateWFR2PRED(c: CompositeStateME) : boolean =
      LET s: finite_set[PseudoStateME]
            = {p: PseudoStateME |
                   (isaStateVertex(p) ∈ subvertices(c)) ∧ kind(p) = deepHistory}
        IN card(s) ≤ 1

  CompositeStateWFR3PRED(c: CompositeStateME) : boolean =
      LET s: finite_set[PseudoStateME]
            = {p: PseudoStateME |
                   (isaStateVertex(p) ∈ subvertices(c)) ∧ kind(p) = shallowHistory}
        IN card(s) ≤ 1

  CompositeStateWFR4PRED(c: CompositeStateME) : boolean =
      LET s: finite_set[StateVertexABS]
            = {v: StateVertexABS |
                   (v ∈ subvertices(c)) ∧
                    isKindOf(isaElement(isaModelElement(v)))(CompositeStateME)}
        IN isConcurrent(c) ⊃ card(s) ≥ 2

  CompositeStateWFR5PRED(c: CompositeStateME) : boolean =
      isConcurrent(c) ⊃
       (∀ (v: StateVertexABS) :
          (v ∈ subvertices(c)) ⊃
           isKindOf(isaElement(isaModelElement(v)))(CompositeStateME))

  CompositeStateWFR6PRED(c: CompositeStateME) : boolean =
      (∀ (v: StateVertexABS) :
         (v ∈ subvertices(c)) ⊃
          card(containerASS(v)) = 1 ∧
           (∃ (c2: CompositeStateME) :
              (c2 ∈ containerASS(v)) ∧ c2 = c))

  CompositeStateWFRLEMMA : LEMMA
    (∀ (c: CompositeStateME) :
       CompositeStateWFR1PRED(c) ∧
```

```
    CompositeStateWFR2PRED(c) ∧
     CompositeStateWFR3PRED(c) ∧
      CompositeStateWFR4PRED(c) ∧
       CompositeStateWFR5PRED(c) ∧ CompositeStateWFR6PRED(c))

FinalStateWFR1PRED(f: FinalStateME): boolean =
    card(outgoingsASS(isaStateVertex(isaState(f)))) = 0

FinalStateWFRLEMMA: LEMMA (∀ (f: FinalStateME): FinalStateWFR1PRED(f))

GuardWFR1PRED(g: GuardME): boolean

GuardWFRLEMMA: LEMMA (∀ (g: GuardME): GuardWFR1PRED(g))

PseudoStateWFR1PRED(p: PseudoStateME): boolean =
    kind(p) = initial ⊃
     card(outgoingsASS(isaStateVertex(p))) ≤ 1 ∧
      empty?(incomingsASS(isaStateVertex(p)))

PseudoStateWFR2PRED(p: PseudoStateME): boolean =
    kind(p) = deepHistory ∨ kind(p) = shallowHistory ⊃
     card(outgoingsASS(isaStateVertex(p))) ≤ 1

PseudoStateWFR3PRED(p: PseudoStateME): boolean =
    kind(p) = join ⊃
     card(outgoingsASS(isaStateVertex(p))) = 1 ∧
      card(incomingsASS(isaStateVertex(p))) ≥ 2

PseudoStateWFR4PRED(p: PseudoStateME): boolean =
    kind(p) = fork ⊃
     card(incomingsASS(isaStateVertex(p))) = 1 ∧
      card(outgoingsASS(isaStateVertex(p))) ≥ 2

PseudoStateWFR5PRED(p: PseudoStateME): boolean =
    kind(p) = junction ⊃
     card(outgoingsASS(isaStateVertex(p))) ≥ 1 ∧
      card(incomingsASS(isaStateVertex(p))) ≥ 1

PseudoStateWFR6PRED(p: PseudoStateME): boolean =
    kind(p) = choice ⊃
     card(incomingsASS(isaStateVertex(p))) ≥ 1 ∧
      card(outgoingsASS(isaStateVertex(p))) ≥ 1

PseudoStateWFRLEMMA: LEMMA
  (∀ (p: PseudoStateME):
```

```
      PseudoStateWFR1PRED(p) ∧
       PseudoStateWFR2PRED(p) ∧
        PseudoStateWFR3PRED(p) ∧
         PseudoStateWFR4PRED(p) ∧
          PseudoStateWFR5PRED(p) ∧ PseudoStateWFR6PRED(p))

StateMachineWFR1PRED(m: StateMachineME): boolean =
    (∃ (e: ModelElementABS):
       (e ∈ contextASS(m)) ⊃
        isKindOf(isElement(e))(BehavioralFeatureABS) ∨
         isKindOf(isElement(e))(ClassifierABS))

StateMachineWFR2PRED(m: StateMachineME): boolean =
    isKindOf(isElement(isModelElement(isStateVertex(top(m)))))
        (CompositeStateME)

StateMachineWFR3PRED(m: StateMachineME): boolean =
    empty?(containerASS(isStateVertex(top(m))))

StateMachineWFR4PRED(m: StateMachineME): boolean =
    empty?(outgoingsASS(isStateVertex(top(m))))

StateMachineWFR5PRED(m: StateMachineME): boolean =
    (∃ (e: ModelElementABS):
       (e ∈ contextASS(m)) ∧ isKindOf(isElement(e))(BehavioralFeatureABS) ⊃
        LET s: finite_set[TransitionME]
               = {t: TransitionME |
                     (t ∈ transitions(m)) ∧
                      ¬ (isKindOf(isElement(isModelElement(sourceASS(t))))
                            (PseudoStateME) ∧
                         (∃ (p: PseudoStateME):
                            isStateVertex(p) = sourceASS(t) ∧ kind(p) = initial))}
          IN (∀ (t: TransitionME): (t ∈ s) ⊃ empty?(trigger(t))))

StateMachineWFRLEMMA: LEMMA
  (∀ (m: StateMachineME):
      StateMachineWFR1PRED(m) ∧
       StateMachineWFR2PRED(m) ∧
        StateMachineWFR3PRED(m) ∧
         StateMachineWFR4PRED(m) ∧ StateMachineWFR5PRED(m))

SynchStateWFR1PRED(s: SynchStateME): boolean =
    bound(s) > 0 ∨ bound(s) = UNLIMITED

SynchStateWFRLEMMA: LEMMA (∀ (s: SynchStateME): SynchStateWFR1PRED(s))
```

```
SubmachineStateWFR1PRED ( s :  SubmachineStateME ) :  boolean  =
     ( ∀  ( v :  StateVertexABS ) :
         ( v ∈ subvertices(isaCompositeState(s)))  ⊃
          isKindOf ( isaElement ( isaModelElement ( v ) ) ) ( StubStateME ) )

SubmachineStateWFR2PRED ( s :  SubmachineStateME ) :  boolean  =
     ¬  isConcurrent ( isaCompositeState ( s ) )

SubmachineStateWFRLEMMA :  LEMMA
   ( ∀  ( s :  SubmachineStateME ) :
      SubmachineStateWFR1PRED ( s )  ∧  SubmachineStateWFR2PRED ( s ) )

TransitionWFR1PRED ( t :  TransitionME ) :  boolean  =
     isKindOf ( isaElement ( isaModelElement ( sourceASS ( t ) ) ) ) ( PseudoStateME )  ⊃
      ( ∃  ( p :  PseudoStateME ) :
          isaStateVertex ( p )  =  sourceASS ( t )  ∧  kind ( p )  =  fork  ⊃
           empty? ( guard ( t ) )  ∧  empty? ( trigger ( t ) ) )

TransitionWFR2PRED ( t :  TransitionME ) :  boolean  =
     isKindOf ( isaElement ( isaModelElement ( targetASS ( t ) ) ) ) ( PseudoStateME )  ⊃
      ( ∃  ( p :  PseudoStateME ) :
          isaStateVertex ( p )  =  targetASS ( t )  ∧  kind ( p )  =  join  ⊃
           empty? ( guard ( t ) )  ∧  empty? ( trigger ( t ) ) )

TransitionWFR3PRED ( t :  TransitionME ) :  boolean  =
     ¬  empty? ( stateMachineASS ( t ) )  ⊃
      isKindOf ( isaElement ( isaModelElement ( sourceASS ( t ) ) ) ) ( PseudoStateME )  ⊃
       ( ∃  ( p :  PseudoStateME ) :
           isaStateVertex ( p )  =  sourceASS ( t )  ∧  kind ( p )  =  fork  ⊃
            isKindOf ( isaElement ( isaModelElement ( targetASS ( t ) ) ) ) ( StateME ) )

TransitionWFR4PRED ( t :  TransitionME ) :  boolean  =
     ¬  empty? ( stateMachineASS ( t ) )  ⊃
      isKindOf ( isaElement ( isaModelElement ( targetASS ( t ) ) ) ) ( PseudoStateME )  ⊃
       ( ∃  ( p :  PseudoStateME ) :
           isaStateVertex ( p )  =  targetASS ( t )  ∧  kind ( p )  =  join  ⊃
            isKindOf ( isaElement ( isaModelElement ( sourceASS ( t ) ) ) ) ( StateME ) )

TransitionWFR5PRED ( t :  TransitionME ) :  boolean  =
     isKindOf ( isaElement ( isaModelElement ( sourceASS ( t ) ) ) ) ( PseudoStateME )  ⊃
      empty? ( trigger ( t ) )

TransitionWFR6PRED ( t :  TransitionME ) :  boolean  =
     isKindOf ( isaElement ( isaModelElement ( targetASS ( t ) ) ) ) ( PseudoStateME )  ⊃
```

```
      (∃ (p: PseudoStateME):
         isStateVertex(p) = targetASS(t) ∧ kind(p) = join ⊃
          (∃ (c: CompositeStateME):
             (c ∈ containerASS(sourceASS(t))) ∧ isConcurrent(c)))

 TransitionWFR7PRED(t: TransitionME): boolean =
     isKindOf(isaElement(isaModelElement(sourceASS(t))))(PseudoStateME) ⊃
      (∃ (p: PseudoStateME):
         isStateVertex(p) = sourceASS(t) ∧ kind(p) = fork ⊃
          (∃ (c: CompositeStateME):
             (c ∈ containerASS(targetASS(t))) ∧ isConcurrent(c)))

 TransitionWFR8PRED(t: TransitionME): boolean =
     isKindOf(isaElement(isaModelElement(sourceASS(t))))(PseudoStateME) ⊃
      ((∃ (p: PseudoStateME): isStateVertex(p) = sourceASS(t) ∧ kind(p) = initial) ⊃
        (empty?(trigger(t)) ∨
          (∃ (c: CompositeStateME, m: StateMachineME, e: EventABS, a: CallEventME):
             (c ∈ containerASS(sourceASS(t))) ∧
              (m ∈ stateMachineASS(t)) ∧
               (e ∈ trigger(t)) ∧
                isaEvent(a) = e ∧ isaState(c) = top(m) ∧
                 stereotypeAUX(a) = createSTE)
           ∨
           (∃ (m: StateMachineME, d: ModelElementABS, e: EventABS, a: CallEventME):
              (m ∈ stateMachineASS(t)) ∧
               (d ∈ contextASS(m)) ∧
                (e ∈ trigger(t)) ∧
                 isaEvent(a) = e ∧
                  isKindOf(isaElement(d))(BehavioralFeatureABS) ∧
                   isKindOf(isaElement(isaModelElement(e)))(CallEventME) ∧
                    isaModelElement(isaFeature(isaBehavioralFeature(operationASS(a)))) =
                     d)))

 TransitionWFRLEMMA: LEMMA
   (∀ (t: TransitionME):
      TransitionWFR1PRED(t) ∧
       TransitionWFR2PRED(t) ∧
        TransitionWFR3PRED(t) ∧
         TransitionWFR4PRED(t) ∧
          TransitionWFR5PRED(t) ∧
           TransitionWFR6PRED(t) ∧
            TransitionWFR7PRED(t) ∧ TransitionWFR8PRED(t))
END stateMachines_wfr
```

B.2.7 UML Package *Use Cases*

```
useCases_abs: THEORY
 BEGIN

  IMPORTING backbone_abs, relationships_abs, instancesAndLinks_abs

  ExtensionPointME: TYPE =
  [# isaModelElement: ModelElementABS, location: LocationReferenceCC #]

  ActorME: TYPE = [# isaClassifier: ClassifierABS #]

  UseCaseME: TYPE = [# isaClassifier: ClassifierABS #]

  UseCaseInstanceME: TYPE = [# isaInstance: InstanceME #]

  IncludeME: TYPE = [# isaRelationship: RelationshipABS #]

  ExtendME: TYPE =
  [# isaRelationship: RelationshipABS, condition: BooleanExpressionCC #]

  includesASS: [UseCaseME → finite_set[IncludeME]]

  extendsASS: [UseCaseME → finite_set[ExtendME]]

  extensionPointsASS: [UseCaseME → finite_set[ExtensionPointME]]

  additionASS: [IncludeME → UseCaseME]

  baseASS: [IncludeME → UseCaseME]

  extensionASS: [ExtendME → UseCaseME]

  baseASS: [ExtendME → UseCaseME]

  extensionPointsASS: [ExtendME → finite_sequence[ExtensionPointME]]
 END useCases_abs
```

```
useCases_aux : THEORY
 BEGIN

  IMPORTING backbone_abs, useCases_abs, core_aux

  specificationPathAUX(u: UseCaseME): finite_set[NameSpaceABS] =
       {n: NameSpaceABS |
            (n ∈ allSurroundingNamespacesAUX(isaNameSpace(isaClassifier(u)))) ∧
             (isKindOf(isaElement(isaModelElement(n)))(SubsystemME) ∨
               isKindOf(isaElement(isaModelElement(n)))(ClassME))}

  allExtensionPointsAUX(u: UseCaseME): finite_set[ExtensionPointME] =
       LET s1: finite_set[ExtensionPointME] = extensionPointsASS(u),
            s2: finite_set[ExtensionPointME]
                = {p: ExtensionPointME |
                        (∃ (u2: UseCaseME):
                            (isaClassifier(u2) ∈ allParentsAUX(isaClassifier(u))) ∧
                             (p ∈ extensionPointsASS(u2)))}
          IN (s1 ∪ s2)
 END useCases_aux
```

```
useCases_wfr : THEORY
 BEGIN

  IMPORTING useCases_aux

  ActorWFR1PRED(a: ActorME): boolean =
      (∀ (c: AssociationME):
          (c ∈ associationsAUX(isClassifier(a))) ⊃
           length(connections(c)) = 2 ∧
            (∃ (e: AssociationEndME):
               (e ∈ allConnectionsAUX(c)) ∧
                 isKindOf(isElement(isModelElement(e)))(ActorME))
             ∧
             (∃ (e: AssociationEndME):
                (e ∈ allConnectionsAUX(c)) ∧
                 (isKindOf(isElement(isModelElement(e)))(UseCaseME) ∨
                   isKindOf(isElement(isModelElement(e)))(SubsystemME) ∨
                    isKindOf(isElement(isModelElement(e)))(ClassME))))

  ActorWFR2PRED(a: ActorME): boolean =
      empty?(contentsAUX(isNameSpace(isClassifier(a))))

  ActorWFRLEMMA: LEMMA
    (∀ (a: ActorME): ActorWFR1PRED(a) ∧ ActorWFR2PRED(a))

  ExtendWFR1PRED(e: ExtendME): boolean =
      (∀ (i: below(length(extensionPointsASS(e)))):
          (nth(finseq2list(extensionPointsASS(e)), i) ∈ allExtensionPointsAUX(baseASS(e))))

  ExtendWFRLEMMA: LEMMA (∀ (e: ExtendME): ExtendWFR1PRED(e))

  ExtensionPointWFR1PRED(p: ExtensionPointME): boolean =
      ¬ body(name(isModelElement(p))) = ""

  ExtensionPointWFRLEMMA: LEMMA
    (∀ (p: ExtensionPointME): ExtensionPointWFR1PRED(p))

  UseCaseWFR1PRED(u: UseCaseME): boolean =
      (∀ (a: AssociationME):
          (a ∈ associationsAUX(isClassifier(u))) ⊃
           length(connections(a)) = 2)

  UseCaseWFR2PRED(u: UseCaseME): boolean =
      (∀ (a: AssociationME):
          (a ∈ associationsAUX(isClassifier(u))) ⊃
```

```
(∀ (e₁, e₂: AssociationEndME):
    (e₁ ∈ allConnectionsAUX(a)) ∧ (e₂ ∈ allConnectionsAUX(a)) ⊃
     (∃ (u₁, u₂: UseCaseME):
         typeASS(e₁) = isaClassifier(u₁) ∧
          typeASS(e₂) = isaClassifier(u₂) ∧
           ((empty?(specificationPathAUX(u₁)) ∧
                   empty?(specificationPathAUX(u₂))) ∨
             (¬ (specificationPathAUX(u₁) ⊆ specificationPathAUX(u₂)) ∧
               ¬ (specificationPathAUX(u₂) ⊆ specificationPathAUX(u₁))))))

UseCaseWFR3PRED(u: UseCaseME): boolean =
    empty?(contentsAUX(isaNameSpace(isaClassifier(u))))

UseCaseWFR4PRED(u: UseCaseME): boolean =
    (∀ (p₁, p₂: ExtensionPointME):
       (p₁ ∈ allExtensionPointsAUX(u)) ∧
        (p₂ ∈ allExtensionPointsAUX(u)) ∧
         name(isaModelElement(p₁)) = name(isaModelElement(p₂))
        ⊃ p₁ = p₂)

UseCaseWFRLEMMA: LEMMA
  (∀ (u: UseCaseME):
     UseCaseWFR1PRED(u) ∧
      UseCaseWFR2PRED(u) ∧ UseCaseWFR3PRED(u) ∧ UseCaseWFR4PRED(u))

UseCaseInstanceWFR1PRED(i: UseCaseInstanceME): boolean =
    (∀ (c: ClassifierABS):
       (c ∈ classifiersASS(isaInstance(i))) ⊃
        isKindOf(isaElement(isaModelElement(isaGeneralizableElement(c))))
             (UseCaseME))

UseCaseInstanceWFRLEMMA: LEMMA
  (∀ (i: UseCaseInstanceME): UseCaseInstanceWFR1PRED(i))
END useCases_wfr
```

Appendix C

Formalization of RTUML Well-formedness Rules in PVS

The RTUML modeling notation consists of a subset of UML diagrams, with a minimal set of extensions and a set of well-formedness rules. The well-formedness rules constrain the core UML notation to capture the description of a real-time reactive system model. Section 4.3.1 gives an informal description of the restrictions on the core UML notation. Section 4.3.2 gives a corresponding description of these constraints in terms of the model elements from the UML abstract syntax, and includes a formalization of these well-formedness rules in OCL. Section D.1 gives the PVS Theories for the RTUML well-formedness rules for each of the UML packages Foundation, Collaborations, and State Machines.

C.1 PVS Theories for RTUML Well-formedness Rules

C.1.1 RTUML Well-formedness Rules for UML Package *Foundation*

```
rtumlFoundation_wfr: THEORY
 BEGIN

  IMPORTING backbone_abs, relationships_abs, dependencies_abs, classifiers_abs,
            auxiliaryElements_abs, core_aux

  RTUMLClassWFR1PRED(c: ClassME): boolean =
      stereotypeAUX(c) = grcSTE ∨ stereotypeAUX(c) = porttypeSTE

  RTUMLClassWFR2PRED(c: ClassME): boolean =
      isRoot(isGeneralizableElement(isClassifier(c))) ∧
       isLeaf(isGeneralizableElement(isClassifier(c))) ∧
        ¬ isAbstract(isGeneralizableElement(isClassifier(c)))

  RTUMLClassWFR3PRED(c: ClassME): boolean =
      (∀ (f: FeatureABS):
         (f ∈ finseq2list(features(isClassifier(c)))) ⊃
          isKindOf(isElement(isModelElement(f)))(AttributeME))

  RTUMLClassWFRLEMMA: LEMMA
    (∀ (c: ClassME):
       RTUMLClassWFR1PRED(c) ∧
        RTUMLClassWFR2PRED(c) ∧ RTUMLClassWFR3PRED(c))

  RTUMLGRCClassWFR1PRED(c: ClassME): boolean =
      stereotypeAUX(c) = grcSTE ⊃ isActive(c)

  RTUMLGRCClassWFR2PRED(c: ClassME): boolean =
      stereotypeAUX(c) = grcSTE ⊃
       (∀ (a: AssociationME):
          (a ∈ associationsAUX(isClassifier(c))) ⊃
           stereotypeAUX(a) = portaggregationSTE)

  RTUMLGRCClassWFR3PRED(c: ClassME): boolean =
      stereotypeAUX(c) = grcSTE ⊃
       card(associationsAUX(isClassifier(c))) ≥ 1

  RTUMLGRCClassWFR4PRED(c: ClassME): boolean =
      stereotypeAUX(c) = grcSTE ⊃
       (∀ (a: AttributeME):
          (a ∈ allAttributesAUX(isClassifier(c))) ⊃
```

targetScope (isaStructuralFeature (a)) = instance)

RTUMLGRCClassWFR5PRED (c: ClassME) : boolean =
stereotypeAUX (c) = grcSTE $\supset$
($\forall$ (a: AttributeME) :
($a \in$ allAttributesAUX(isaClassifier(c))) $\supset$
max (multiplicity (isaStructuralFeature (a))) = 1)

RTUMLGRCClassWFR6PRED (c: ClassME) : boolean =
stereotypeAUX (c) = grcSTE $\supset$
($\forall$ (a: AttributeME) :
($a \in$ allAttributesAUX(isaClassifier(c))) $\supset$
visibility (isaFeature (isaStructuralFeature (a))) = private)

RTUMLGRCClassWFR7PRED (c: ClassME) : boolean =
stereotypeAUX (c) = grcSTE $\supset$
($\forall$ (a: AttributeME) :
($a \in$ allAttributesAUX(isaClassifier(c))) $\supset$
changeability (isaStructuralFeature (a)) = none)

RTUMLGRCClassWFR8PRED (c: ClassME) : boolean =
stereotypeAUX (c) = grcSTE $\supset$
($\forall$ (a: AttributeME) :
($a \in$ allAttributesAUX(isaClassifier(c))) $\supset$
(($\exists$ (c_2: ClassME) :
typeASS (isaStructuralFeature (a)) = isaClassifier (c_2) $\wedge$
stereotypeAUX (c_2) = porttypeSTE)
$\vee$
($\exists$ (d: DataTypeME) :
typeASS (isaStructuralFeature (a)) = isaClassifier (d)) $\vee$
($\exists$ (b: BindingME, d: DataTypeME, c_1, c_2: ClassifierABS) :
card (clientsASS (isaDependency (b))) = 1 $\wedge$
card (suppliersASS (isaDependency (b))) = 1 $\wedge$
(isaModelElement(isaGeneralizableElement(c_1)) $\in$
clientsASS(isaDependency(b))) $\wedge$
(isaModelElement(isaGeneralizableElement(c_2)) $\in$
suppliersASS(isaDependency(b))) $\wedge$
typeASS (isaStructuralFeature (a)) = c_1 $\wedge$
c_2 = isaClassifier (d))))

RTUMLGRCClassWFR9PRED (c: ClassME) : boolean =
stereotypeAUX (c) = grcSTE $\supset$
($\forall$ (a: AttributeME) :
($a \in$ allAttributesAUX(isaClassifier(c))) $\wedge$
($\exists$ (c_2: ClassME) :

```
                typeASS(isaStructuralFeature(a)) = isaClassifier(c2) ∧
                 stereotypeAUX(c2) = porttypeSTE
                 ⊃
                 (∃ (a: AssociationME, e1, e2: AssociationEndME):
                    stereotypeAUX(a) = portaggregationSTE ∧
                     e1 ≠ e2 ∧
                      (e1 ∈ finseq2list(connections(a))) ∧
                       (e2 ∈ finseq2list(connections(a))) ∧
                        typeASS(e1) = isaClassifier(c) ∧
                         typeASS(e2) = isaClassifier(c2))))

RTUMLGRCClassWFR10PRED(c: ClassME): boolean =
    stereotypeAUX(c) = grcSTE ⊃
     (∀ (a: AttributeME):
        (a ∈ allAttributesAUX(isaClassifier(c))) ∧
         (∃ (b: BindingME, l: ClassifierABS):
            (isaModelElement(isaGeneralizableElement(l)) ∈ clientsASS(isaDependency(b))) ∧
             typeASS(isaStructuralFeature(a)) = l
             ⊃
             (∀ (m: ModelElementABS):
                (m ∈ finseq2list(arguments(b))) ⊃
                 ((∃ (d: DataTypeME):
                     isaModelElement(isaGeneralizableElement(isaClassifier(d))) = m)
                   ∨
                   (∃ (c: ClassME):
                      stereotypeAUX(c) = porttypeSTE ∧
                       isaModelElement(isaGeneralizableElement(isaClassifier(c))) =
                        m)))))

RTUMLGRCClassWFR11PRED(c: ClassME): boolean =
    stereotypeAUX(c) = grcSTE ⊃
     (∀ (a: AttributeME):
        (a ∈ allAttributesAUX(isaClassifier(c))) ∧
         (∃ (b: BindingME, l: ClassifierABS):
            (isaModelElement(isaGeneralizableElement(l)) ∈ clientsASS(isaDependency(b))) ∧
             typeASS(isaStructuralFeature(a)) = l ∧
              (∃ (m: ModelElementABS, c2: ClassME):
                 (m ∈ finseq2list(arguments(b))) ∧
                  stereotypeAUX(c2) = porttypeSTE ∧
                   isaModelElement(isaGeneralizableElement(isaClassifier(c2))) = m
                  ⊃
                  (∃ (a: AssociationME, e1, e2: AssociationEndME):
                     stereotypeAUX(a) = portaggregationSTE ∧
                      e1 ≠ e2 ∧
                       (e1 ∈ finseq2list(connections(a))) ∧
```

$(e_2 \in \text{finseq2list}(\text{connections}(a))) \wedge$
$\text{typeASS}(e_1) = \text{isaClassifier}(c) \wedge$
$\text{typeASS}(e_2) = \text{isaClassifier}(c_2)))))$

RTUMLGRCClassWFRLEMMA: LEMMA
$(\forall\ (c\text{: ClassME}):$
$\text{stereotypeAUX}(c) = \text{grcSTE} \supset$
$\text{RTUMLGRCClassWFR1PRED}(c) \wedge$
$\text{RTUMLGRCClassWFR2PRED}(c) \wedge$
$\text{RTUMLGRCClassWFR3PRED}(c) \wedge$
$\text{RTUMLGRCClassWFR4PRED}(c) \wedge$
$\text{RTUMLGRCClassWFR5PRED}(c) \wedge$
$\text{RTUMLGRCClassWFR6PRED}(c) \wedge$
$\text{RTUMLGRCClassWFR7PRED}(c) \wedge$
$\text{RTUMLGRCClassWFR8PRED}(c) \wedge$
$\text{RTUMLGRCClassWFR9PRED}(c) \wedge$
$\text{RTUMLGRCClassWFR10PRED}(c) \wedge \text{RTUMLGRCClassWFR11PRED}(c))$

$\text{RTUMLPortTypeClassWFR1PRED}(c\text{: ClassME})\text{: boolean} =$
$\text{stereotypeAUX}(c) = \text{porttypeSTE} \supset \neg\ \text{isActive}(c)$

$\text{RTUMLPortTypeClassWFR2PRED}(c\text{: ClassME})\text{: boolean} =$
$\text{stereotypeAUX}(c) = \text{porttypeSTE} \supset$
$(\exists\ (a\text{: AssociationME}, e_1, e_2\text{: AssociationEndME}, c_2\text{: ClassME}):$
$\text{stereotypeAUX}(a) = \text{portaggregationSTE} \wedge$
$e_1 \neq e_2 \wedge$
$(e_1 \in \text{finseq2list}(\text{connections}(a))) \wedge$
$(e_2 \in \text{finseq2list}(\text{connections}(a))) \wedge$
$\text{stereotypeAUX}(c_2) = \text{grcSTE} \wedge$
$\text{typeASS}(e_1) = \text{isaClassifier}(c) \wedge$
$\text{typeASS}(e_2) = \text{isaClassifier}(c_2) \wedge$
$(\forall\ (a_2\text{: AssociationME}):$
$\text{stereotypeAUX}(a_2) = \text{portaggregationSTE} \wedge a \neq a_2 \supset$
$(\forall\ (e_3\text{: AssociationEndME}):$
$(e_3 \in \text{finseq2list}(\text{connections}(a))) \supset$
$\text{typeASS}(e_3) \neq \text{isaClassifier}(c))))$

$\text{RTUMLPortTypeClassWFR3PRED}(c\text{: ClassME})\text{: boolean} =$
$\text{stereotypeAUX}(c) = \text{porttypeSTE} \supset$
$(\exists\ (a\text{: AssociationME}, e_1, e_2\text{: AssociationEndME}, c_2\text{: ClassME}):$
$\text{stereotypeAUX}(a) = \text{portlinkSTE} \wedge$
$e_1 \neq e_2 \wedge$
$(e_1 \in \text{finseq2list}(\text{connections}(a))) \wedge$
$(e_2 \in \text{finseq2list}(\text{connections}(a))) \wedge$
$\text{stereotypeAUX}(c_2) = \text{porttypeSTE} \wedge$

typeASS (e_1) = isaClassifier (c) ∧
typeASS (e_2) = isaClassifier (c_2))

RTUMLPortTypeClassWFR4PRED (c : ClassME) : boolean =
stereotypeAUX (c) = porttypeSTE ⊃
length (features (isaClassifier (c))) = 1

RTUMLPortTypeClassWFR5PRED (c : ClassME) : boolean =
stereotypeAUX (c) = porttypeSTE ⊃
(∀ (a : AttributeME) :
(a ∈ allAttributesAUX(isaClassifier(c))) ⊃
targetScope (isaStructuralFeature (a)) = instance)

RTUMLPortTypeClassWFR6PRED (c : ClassME) : boolean =
stereotypeAUX (c) = porttypeSTE ⊃
(∀ (a : AttributeME) :
(a ∈ allAttributesAUX(isaClassifier(c))) ⊃
max (multiplicity (isaStructuralFeature (a))) = 1)

RTUMLPortTypeClassWFR7PRED (c : ClassME) : boolean =
stereotypeAUX (c) = porttypeSTE ⊃
(∀ (a : AttributeME) :
(a ∈ allAttributesAUX(isaClassifier(c))) ⊃
visibility (isaFeature (isaStructuralFeature (a))) = public)

RTUMLPortTypeClassWFR8PRED (c : ClassME) : boolean =
stereotypeAUX (c) = porttypeSTE ⊃
(∀ (f : StructuralFeatureABS) :
(isaFeature(f) ∈ allFeaturesAUX(isaClassifier(c))) ⊃
(∃ (l : ClassifierABS) :
typeASS (f) = l ∧
changeability (f) = frozen ∧
body (name (isaModelElement (isaFeature (f)))) = `"events"` ∧
body (name (isaModelElement (isaGeneralizableElement (l)))) =
`"Set"`))

RTUMLPortTypeClassWFRLEMMA : LEMMA
(∀ (c : ClassME) :
stereotypeAUX (c) = porttypeSTE ⊃
RTUMLPortTypeClassWFR1PRED (c) ∧
RTUMLPortTypeClassWFR2PRED (c) ∧
RTUMLPortTypeClassWFR3PRED (c) ∧
RTUMLPortTypeClassWFR4PRED (c) ∧
RTUMLPortTypeClassWFR5PRED (c) ∧
RTUMLPortTypeClassWFR6PRED (c) ∧

```
            RTUMLPortTypeClassWFR7PRED(c) ∧
             RTUMLPortTypeClassWFR8PRED(c))

RTUMLAssociationWFR1PRED(a: AssociationME): boolean =
    stereotypeAUX(a) = portaggregationSTE ∨
     stereotypeAUX(a) = portlinkSTE

RTUMLAssociationWFR2PRED(a: AssociationME): boolean =
    length(connections(a)) = 2

RTUMLAssociationWFR3PRED(a: AssociationME): boolean =
    (∀ (e: AssociationEndME):
       (e ∈ finseq2list(connections(a))) ⊃ isNavigable(e))

RTUMLAssociationWFR4PRED(a: AssociationME): boolean =
    (∀ (e: AssociationEndME):
       (e ∈ finseq2list(connections(a))) ⊃ ordering(e) = unordered)

RTUMLAssociationWFR5PRED(a: AssociationME): boolean =
    (∀ (e: AssociationEndME):
       (e ∈ finseq2list(connections(a))) ⊃
        targetScope(e) = instance)

RTUMLAssociationWFR6PRED(a: AssociationME): boolean =
    (∀ (e: AssociationEndME):
       (e ∈ finseq2list(connections(a))) ⊃ max(multiplicity(e)) = 1)

RTUMLAssociationWFR7PRED(a: AssociationME): boolean =
    (∀ (e: AssociationEndME):
       (e ∈ finseq2list(connections(a))) ⊃ changeability(e) = none)

RTUMLAssociationWFR8PRED(a: AssociationME): boolean =
    (∀ (e: AssociationEndME):
       (e ∈ finseq2list(connections(a))) ⊃ visibility(e) = public)

RTUMLAssociationWFRLEMMA: LEMMA
  (∀ (a: AssociationME):
     RTUMLAssociationWFR1PRED(a) ∧
      RTUMLAssociationWFR2PRED(a) ∧
       RTUMLAssociationWFR3PRED(a) ∧
        RTUMLAssociationWFR4PRED(a) ∧
         RTUMLAssociationWFR5PRED(a) ∧
          RTUMLAssociationWFR6PRED(a) ∧
           RTUMLAssociationWFR7PRED(a) ∧ RTUMLAssociationWFR8PRED(a))
```

RTUMLPortAggregationAssociationWFR1PRED(a: AssociationME) : boolean =
  stereotypeAUX(a) = portaggregationSTE $\supset$
   ($\exists$ (e_1, e_2: AssociationEndME, c_1, c_2: ClassME) :
    $e_1 \neq e_2$ $\wedge$
     $c_1 \neq c_2$ $\wedge$
      ($e_1 \in$ finseq2list(connections(a))) $\wedge$
       ($e_2 \in$ finseq2list(connections(a))) $\wedge$
        aggregation(e_1) = composite $\wedge$
         aggregation(e_2) = none $\wedge$
          typeASS(e_1) = isaClassifier(c_1) $\wedge$
           typeASS(e_2) = isaClassifier(c_2) $\wedge$
            stereotypeAUX(c_1) = grcSTE $\wedge$
             stereotypeAUX(c_2) = porttypeSTE)

RTUMLPortAggregationAssociationWFRLEMMA: LEMMA
 ($\forall$ (a: AssociationME) :
   stereotypeAUX(a) = portaggregationSTE $\supset$
    RTUMLPortAggregationAssociationWFR1PRED(a))

RTUMLPortLinkAssociationWFR1PRED(a: AssociationME) : boolean =
  stereotypeAUX(a) = portlinkSTE $\supset$
   ($\exists$ (e_1, e_2: AssociationEndME, c_1, c_2: ClassME) :
    $e_1 \neq e_2$ $\wedge$
     $c_1 \neq c_2$ $\wedge$
      ($e_1 \in$ finseq2list(connections(a))) $\wedge$
       ($e_2 \in$ finseq2list(connections(a))) $\wedge$
        aggregation(e_1) = none $\wedge$
         aggregation(e_2) = none $\wedge$
          typeASS(e_1) = isaClassifier(c_1) $\wedge$
           typeASS(e_2) = isaClassifier(c_2) $\wedge$
            stereotypeAUX(c_1) = porttypeSTE $\wedge$
             stereotypeAUX(c_2) = porttypeSTE)

RTUMLPortLinkAssociationWFRLEMMA: LEMMA
 ($\forall$ (a: AssociationME) :
   stereotypeAUX(a) = portlinkSTE $\supset$
    RTUMLPortLinkAssociationWFR1PRED(a))
END rtumlFoundation_wfr

C.1.2 RTUML Well-formedness Rules for UML Package *Collaborations*

```
rtumlCollaborations_wfr: THEORY
 BEGIN

  IMPORTING collaborations_abs

  RTUMLCollaborationWFR1PRED(c: CollaborationME): boolean =
      (∀ (m: ModelElementABS):
          (m ∈ ownedElements(isaNameSpace(c))) ⊃
           isKindOf(isaElement(m))(ClassifierRoleME) ∨
            isKindOf(isaElement(m))(AssociationRoleME))

  RTUMLCollaborationWFR2PRED(c: CollaborationME): boolean =
      (∀ (r: ClassifierRoleME):
          (isaModelElement(isaGeneralizableElement(isaClassifier(r))) ∈
              ownedElements(isaNameSpace(c)))
           ∧ (∃ (c2: ClassME): baseASS(r) = isClassifier(c2) ∧
                   stereotypeAUX(c2) = grcSTE)
           ⊃
           LET s: finite_set[AssociationRoleME]
                   = {ar: AssociationRoleME |
                          (isaModelElement(isaRelationship(isaAssociation(ar))) ∈
                              ownedElements(isaNameSpace(c)))
                           ∧
                           stereotypeAUX(baseASS(ar)) = portaggregationSTE ∧
                            (∃ (e: AssociationEndRoleME):
                               (e ∈ finseq2list(connections(ar))) ∧ typeASS(e) = r)}
             IN card(s) ≥ 1)

  RTUMLCollaborationWFR3PRED(c: CollaborationME): boolean =
      (∀ (r: ClassifierRoleME):
          (isaModelElement(isaGeneralizableElement(isaClassifier(r))) ∈
              ownedElements(isaNameSpace(c)))
           ∧ (∃ (c2: ClassME): baseASS(r) = isaClassifier(c2) ∧
                 stereotypeAUX(c2) = grcSTE)
           ⊃
           LET s: finite_set[AssociationRoleME]
                   = {a: AssociationRoleME |
                          (isaModelElement(isaRelationship(isaAssociation(a))) ∈
                             ownedElements(isaNameSpace(c))) ∧
                           stereotypeAUX(baseASS(a)) = portaggregationSTE ∧
                            (∃ (e1, e2, e3: AssociationEndRoleME, a2: AssociationRoleME,
                                r2: ClassifierRoleME):
                               (e1 ∈ finseq2list(connections(a))) ∧
```

```
                                    (e₂ ∈ finseq2list(connections(a)))  ∧
                                     typeASS(e₁) = r ∧
                                      typeASS(e₂) = r₂ ∧
                                       (isaModelElement(isaRelationship(isaAssociation(a₂))) ∈
                                          ownedElements(isaNameSpace(c)))
                                        ∧
                                        stereotypeAUX(baseASS(a₂)) = portlinkSTE ∧
                                         (e₃ ∈ finseq2list(connections(a₂))) ∧ typeASS(e₃) = r₂)}
          IN card(s) ≥ 1)

RTUMLCollaborationWFR4PRED(c: CollaborationME): boolean =
    (∀ (r: ClassifierRoleME):
       (isaModelElement(isaGeneralizableElement(isaClassifier(r))) ∈
            ownedElements(isaNameSpace(c)))
        ∧ (∃ (c₂: ClassME): baseASS(r) = isaClassifier(c₂) ∧
               stereotypeAUX(c₂) = porttypeSTE)
        ⊃
        LET s: finite_set[AssociationRoleME]
               = {a: AssociationRoleME |
                     (isaModelElement(isaRelationship(isaAssociation(a))) ∈
                        ownedElements(isaNameSpace(c))) ∧
                      stereotypeAUX(baseASS(a)) = portaggregationSTE ∧
                       (∃ (e: AssociationEndRoleME):
                          (e ∈ finseq2list(connections(a))) ∧ typeASS(e) = r)}
          IN card(s) = 1)

RTUMLCollaborationWFR5PRED(c: CollaborationME): boolean =
    (∀ (r: ClassifierRoleME):
       (isaModelElement(isaGeneralizableElement(isaClassifier(r))) ∈
            ownedElements(isaNameSpace(c)))
        ∧ (∃ (c₂: ClassME): baseASS(r) = isaClassifier(c₂) ∧
               stereotypeAUX(c₂) = porttypeSTE)
        ⊃
        LET s: finite_set[AssociationRoleME]
               = {a: AssociationRoleME |
                     (isaModelElement(isaRelationship(isaAssociation(a))) ∈
                          ownedElements(isaNameSpace(c))) ∧
                      stereotypeAUX(baseASS(a)) = portlinkSTE ∧
                       (∃ (e: AssociationEndRoleME):
                          (e ∈ finseq2list(connections(a))) ∧ typeASS(e) = r)}
          IN card(s) ≤ 1)

RTUMLCollaborationWFRLEMMA: LEMMA
   (∀ (c: CollaborationME):
      RTUMLCollaborationWFR1PRED(c) ∧
```

RTUMLCollaborationWFR2PRED (c) $\wedge$
RTUMLCollaborationWFR3PRED (c) $\wedge$
RTUMLCollaborationWFR4PRED (c) $\wedge$ RTUMLCollaborationWFR5PRED (c))

RTUMLClassifierRoleWFR1PRED (r : ClassifierRoleME) : boolean =
max (multiplicity (r)) = 1

RTUMLClassifierRoleWFR2PRED (r : ClassifierRoleME) : boolean =
($\exists$ (c_2 : ClassME) :
baseASS (r) = isaClassifier (c_2) $\wedge$
(stereotypeAUX (c_2) = grcSTE $\vee$ stereotypeAUX (c_2) = porttypeSTE))

RTUMLClassifierRoleWFRLEMMA : LEMMA
($\forall$ (r : ClassifierRoleME) :
RTUMLClassifierRoleWFR1PRED (r) $\wedge$ RTUMLClassifierRoleWFR2PRED (r))

RTUMLAssociationRoleWFR1PRED (r : AssociationRoleME) : boolean =
max (multiplicity (r)) = 1

RTUMLAssociationRoleWFR2PRED (r : AssociationRoleME) : boolean =
($\exists$ (a : AssociationME) :
baseASS (r) = a $\wedge$
(stereotypeAUX (a) = portaggregationSTE $\vee$
stereotypeAUX (a) = portlinkSTE))

RTUMLAssociationRoleWFR3PRED (r : AssociationRoleME) : boolean =
length (connections (r)) = 2

RTUMLAssociationRoleWFR4PRED (r : AssociationRoleME) : boolean =
($\exists$ (a : AssociationME) :
baseASS (r) = a $\wedge$ stereotypeAUX (a) = portaggregationSTE $\supset$
($\exists$ (e_1 , e_2 : AssociationEndRoleME , r_1 , r_2 : ClassifierRoleME , c_1 , c_2 : ClassME ,
n_1 , n_2 : AssociationEndME) :
$e_1 \neq e_2$ $\wedge$
$r_1 \neq r_2$ $\wedge$
$c_1 \neq c_2$ $\wedge$
$n_1 \neq n_2$ $\wedge$
($e_1 \in$ finseq2list(connections(r))) $\wedge$
($e_2 \in$ finseq2list(connections(r))) $\wedge$
typeASS (e_1) = r_1 $\wedge$
typeASS (e_2) = r_2 $\wedge$
baseASS (r_1) = isaClassifier (c_1) $\wedge$
baseASS (r_2) = isaClassifier (c_2) $\wedge$
stereotypeAUX (c_1) = grcSTE $\wedge$
stereotypeAUX (c_2) = porttypeSTE $\wedge$

```
                    (n1 ∈ finseq2list(connections(a))) ∧
                     (n2 ∈ finseq2list(connections(a))) ∧
                      typeASS(n1) = isClassifier(c1) ∧
                       typeASS(n2) = isClassifier(c2)))

RTUMLAssociationRoleWFR5PRED(r: AssociationRoleME): boolean =
  (∃ (a: AssociationME):
     baseASS(r) = a ∧ stereotypeAUX(a) = portlinkSTE ⊃
      (∃ (e1, e2: AssociationEndRoleME, r1, r2: ClassifierRoleME, c1, c2: ClassME,
          n1, n2: AssociationEndME):
         e1 ≠ e2 ∧
          r1 ≠ r2 ∧
           c1 ≠ c2 ∧
            n1 ≠ n2 ∧
             (e1 ∈ finseq2list(connections(r))) ∧
              (e2 ∈ finseq2list(connections(r))) ∧
               typeASS(e1) = r1 ∧
                typeASS(e2) = r2 ∧
                 baseASS(r1) = isClassifier(c1) ∧
                  baseASS(r2) = isClassifier(c2) ∧
                   stereotypeAUX(c1) = porttypeSTE ∧
                    stereotypeAUX(c2) = porttypeSTE ∧
                     (n1 ∈ finseq2list(connections(a))) ∧
                      (n2 ∈ finseq2list(connections(a))) ∧
                       typeASS(n1) = isClassifier(c1) ∧
                        typeASS(n2) = isClassifier(c2)))

RTUMLAssociationRoleWFRLEMMA: LEMMA
  (∀ (r: AssociationRoleME):
     RTUMLAssociationRoleWFR1PRED(r) ∧
      RTUMLAssociationRoleWFR2PRED(r) ∧
       RTUMLAssociationRoleWFR3PRED(r) ∧
        RTUMLAssociationRoleWFR4PRED(r) ∧
         RTUMLAssociationRoleWFR5PRED(r))

RTUMLInteractionWFR1PRED(i: InteractionME): boolean =
  (∀ (m: MessageME):
     (m ∈ messages(i)) ⊃
      (∃ (r: ClassifierRoleME, c: ClassME):
         senderASS(m) = r ∧
          baseASS(r) = isClassifier(c) ∧
           stereotypeAUX(c) = grcSTE ∧
            (isModelElement(isGeneralizableElement(isClassifier(r))) ∈
                 ownedElements(isNameSpace(contextASS(i))))))
```

```
RTUMLInteractionWFR2PRED(i: InteractionME): boolean =
  (∀ (m: MessageME):
    (m ∈ messages(i)) ⊃
     (∃ (r: ClassifierRoleME, c: ClassME):
        receiverASS(m) = r ∧
         baseASS(r) = isClassifier(c) ∧
          stereotypeAUX(c) = grcSTE ∧
           (isModelElement(isGeneralizableElement(isClassifier(r))) ∈
                ownedElements(isNameSpace(contextASS(i))))))

RTUMLInteractionWFR3PRED(i: InteractionME): boolean =
  (∀ (m: MessageME):
    (m ∈ messages(i)) ⊃
     (∃ (r: AssociationRoleME, a: AssociationME):
        communicationConnectionASS(m) = r ∧
         baseASS(r) = a ∧
          stereotypeAUX(a) = portlinkSTE ∧
           (∃ (n1, n2: AssociationEndME, c1, c2: ClassME):
              n1 ≠ n2 ∧ c1 ≠ c2 ∧
                (n1 ∈ finseq2list(connections(a))) ∧
                 (n2 ∈ finseq2list(connections(a))) ∧
                  typeASS(n1) = isClassifier(c1) ∧
                   typeASS(n2) = isClassifier(c2) ∧
                    stereotypeAUX(c1) = porttypeSTE ∧
                     stereotypeAUX(c2) = porttypeSTE ∧
                      (∃ (c3, c4: ClassME, a2, a3: AssociationME,
                           n3, n4, n5, n6: AssociationEndME):
                         n3 ≠ n4 ∧ n5 ≠ n6 ∧
                           stereotypeAUX(c3) = grcSTE ∧
                            stereotypeAUX(c4) = grcSTE ∧
                             stereotypeAUX(a2) = portaggregationSTE ∧
                              stereotypeAUX(a3) = portaggregationSTE ∧
                               (n3 ∈ finseq2list(connections(a2))) ∧
                                (n4 ∈ finseq2list(connections(a2))) ∧
                                 (n5 ∈ finseq2list(connections(a3))) ∧
                                  (n6 ∈ finseq2list(connections(a3))) ∧
                                   typeASS(n3) = isClassifier(c1) ∧
                                    typeASS(n4) = isClassifier(c3) ∧
                                     typeASS(n5) = isClassifier(c2) ∧
                                      typeASS(n6) = isClassifier(c4) ∧
                                       baseASS(senderASS(m)) =
                                        isClassifier(c3) ∧
                                        baseASS(receiverASS(m)) =
                                         isClassifier(c4)))))
```

```
RTUMLInteractionWFRLEMMA: LEMMA
  (∀ (i: InteractionME):
     RTUMLInteractionWFR1PRED(i) ∧
      RTUMLInteractionWFR2PRED(i) ∧ RTUMLInteractionWFR3PRED(i))
END rtumlCollaborations_wfr
```

C.1.3 RTUML Well-formedness Rules for UML Package *State Machines*

```
rtumlStateMachines_wfr: THEORY
 BEGIN

  IMPORTING stateMachines_abs

  RTUMLStateMachineWFR1PRED(m: StateMachineME): boolean =
      card(contextASS(m)) = 1 ∧
       (∃ (c: ClassME):
          stereotypeAUX(c) = grcSTE ∧
           (isaModelElement(isaGeneralizableElement(isaClassifier(c))) ∈ contextASS(m)))

  RTUMLStateMachineWFRLEMMA: LEMMA
    (∀ (m: StateMachineME): RTUMLStateMachineWFR1PRED(m))

  RTUMLTransitionWFR1PRED(t: TransitionME): boolean =
      isKindOf(isaElement(isaModelElement(sourceASS(t))))(StateME) ∧
       isKindOf(isaElement(isaModelElement(targetASS(t))))(StateME)

  RTUMLTransitionWFR2PRED(t: TransitionME): boolean = card(trigger(t)) = 1

  RTUMLTransitionWFR3PRED(t: TransitionME): boolean = card(guard(t)) = 1

  RTUMLTransitionWFR4PRED(t: TransitionME): boolean = card(effect(t)) = 1

  RTUMLTransitionWFR5PRED(t: TransitionME): boolean =
      (∃ (e: EventABS):
         (e ∈ trigger(t)) ∧
          isKindOf(isaElement(isaModelElement(e)))(SignalEventME))

  RTUMLTransitionWFR6PRED(t: TransitionME): boolean =
      (∃ (e: EventABS): (e ∈ trigger(t)) ∧ length(parameters(e)) = 0)

  RTUMLTransitionWFR7PRED(t: TransitionME): boolean =
      (∃ (a: ActionABS): (a ∈ effect(t)) ∧ ¬ isAsynchronous(a))

  RTUMLTransitionWFRLEMMA: LEMMA
    (∀ (t: TransitionME):
       RTUMLTransitionWFR1PRED(t) ∧
        RTUMLTransitionWFR2PRED(t) ∧
         RTUMLTransitionWFR3PRED(t) ∧
          RTUMLTransitionWFR4PRED(t) ∧
           RTUMLTransitionWFR5PRED(t) ∧
            RTUMLTransitionWFR6PRED(t) ∧ RTUMLTransitionWFR7PRED(t))
```

```
RTUMLStateWFR1PRED(s: StateME): boolean =
    isKindOf(isaElement(isaModelElement(isaStateVertex(s))))(SimpleStateME) ∨
     isKindOf(isaElement(isaModelElement(isaStateVertex(s))))
         (CompositeStateME)

RTUMLStateWFR2PRED(s: StateME): boolean = empty?(exit(s))

RTUMLStateWFR3PRED(s: StateME): boolean = empty?(doActivity(s))

RTUMLStateWFR4PRED(s: StateME): boolean = empty?(deferrableEventsASS(s))

RTUMLStateWFRLEMMA: LEMMA
  (∀ (s: StateME):
     RTUMLStateWFR1PRED(s) ∧
      RTUMLStateWFR2PRED(s) ∧
       RTUMLStateWFR3PRED(s) ∧ RTUMLStateWFR4PRED(s))

RTUMLCompositeStateWFR1PRED(c: CompositeStateME): boolean =
    ¬ isConcurrent(c)

RTUMLCompositeStateWFR2PRED(c: CompositeStateME): boolean =
    (∀ (v: StateVertexABS):
       (v ∈ subvertices(c)) ⊃
        isKindOf(isaElement(isaModelElement(v)))(StateME))

RTUMLCompositeStateWFRLEMMA: LEMMA
  (∀ (c: CompositeStateME):
     RTUMLCompositeStateWFR1PRED(c) ∧ RTUMLCompositeStateWFR2PRED(c))
END rtumlStateMachines_wfr
```

Appendix D

Formalization of Semantic Domain in PVS

The semantic domain defines the abstract model of a real-time reactive system. The domain includes GRC classes, extended statecharts, GRC types, port types, data types, configurations, and scenarios. A GRC type corresponds to a GRC class coupled with an extended statechart. A reactive system model comprises of a set of GRC types, a set of configurations, and a set of scenarios. The core of the semantic domain includes definitions for the time domain, the domain of events, and the domain of states. Section D.1 gives the PVS Theories for the semantic domain.

D.1 PVS Theories for Semantic Domain

D.1.1 Core Semantic Domain Concepts

```
core_sd: THEORY
 BEGIN

  TimeSDC: TYPE = nat

  PortSDC: TYPE

  PortTypeSDC: TYPE = finite_set[PortSDC]

  DataTypeSDC: TYPE

  PortTypeAttributeSDC: TYPE = [# attribute_type: PortTypeSDC #]

  DataTypeAttributeSDC: TYPE = [# attribute_type: DataTypeSDC #]

  NULLPORT_p0: PortSDC

  NULLPORTTYPE_P0: PortTypeSDC = singleton(NULLPORT_p0)

  UNIVERSALSET_DATATYPE: finite_set[DataTypeSDC]

  UNIVERSALSET_PORTTYPE: finite_set[PortTypeSDC]
 END core_sd
```

D.1.2 Semantic Domain Concept *Event*

```
event_sd : THEORY
 BEGIN

  EventSDC : TYPE

  UNIVERSALSET_EVENT : finite_set [ EventSDC ]

  UNIVERSALSET_INTERNALEVENT : finite_set [ EventSDC ]

  UNIVERSALSET_EXTERNALEVENT : finite_set [ EventSDC ]

  UNIVERSALSET_INPUTEVENT : finite_set [ EventSDC ]

  UNIVERSALSET_OUTPUTEVENT : finite_set [ EventSDC ]

  UniversalSets_AX1 : AXIOM
    (UNIVERSALSET_INTERNALEVENT ∩ UNIVERSALSET_EXTERNALEVENT) = ∅

  UniversalSets_AX2 : AXIOM
    (UNIVERSALSET_INTERNALEVENT ∩ UNIVERSALSET_INPUTEVENT) = ∅

  UniversalSets_AX3 : AXIOM
    (UNIVERSALSET_INTERNALEVENT ∩ UNIVERSALSET_OUTPUTEVENT) = ∅

  UniversalSets_AX4 : AXIOM
    (UNIVERSALSET_EXTERNALEVENT ∩ UNIVERSALSET_INPUTEVENT) = ∅

  UniversalSets_AX5 : AXIOM
    (UNIVERSALSET_EXTERNALEVENT ∩ UNIVERSALSET_OUTPUTEVENT) = ∅

  UniversalSets_AX6 : AXIOM
    (UNIVERSALSET_INPUTEVENT ∩ UNIVERSALSET_OUTPUTEVENT) = ∅

  UniversalSets_AX7 : AXIOM
    UNIVERSALSET_EVENT =
     (UNIVERSALSET_INTERNALEVENT ∪
         (UNIVERSALSET_INPUTEVENT ∪ UNIVERSALSET_OUTPUTEVENT))

  input2external :
  [finite_set [ (UNIVERSALSET_INPUTEVENT) ] →
     finite_set [ (UNIVERSALSET_EXTERNALEVENT) ] ]

  input2external_AX1 : AXIOM bijective? ( input2external )
```

```
  output2external :
  [ finite_set [ ( UNIVERSALSET_OUTPUTEVENT ) ]  →
      finite_set [ ( UNIVERSALSET_EXTERNALEVENT ) ] ]

  output2external_AX1 :  AXIOM  bijective? ( output2external )

  conjugate_event :
  [ ( UNIVERSALSET_INPUTEVENT )  →  ( UNIVERSALSET_OUTPUTEVENT ) ]

  conjugate_event_AX1 :  AXIOM  bijective? ( conjugate_event )
END  event_sd
```

D.1.3 Semantic Domain Concept *State*

state_sd : THEORY
BEGIN

StateSDC : TYPE

UNIVERSALSET_STATE : finite_set [StateSDC]

UNIVERSALSET_SIMPLESTATE : finite_set [StateSDC]

UNIVERSALSET_COMPLEXSTATE : finite_set [StateSDC]

universalSets_AX1 : AXIOM
(UNIVERSALSET_SIMPLESTATE $\cap$ UNIVERSALSET_COMPLEXSTATE) = $\emptyset$

universalSets_AX2 : AXIOM
UNIVERSALSET_STATE =
(UNIVERSALSET_SIMPLESTATE $\cup$ UNIVERSALSET_COMPLEXSTATE)

s : VAR StateSDC

c : VAR (UNIVERSALSET_COMPLEXSTATE)

substates : [StateSDC $\rightarrow$ finite_set [StateSDC]]

substates_AX1 : AXIOM
($s \in$ UNIVERSALSET_SIMPLESTATE) $\supset$ substates(s) = $\emptyset$

substates_AX2 : AXIOM
($s \in$ UNIVERSALSET_COMPLEXSTATE) $\supset$ substates(s) $\neq \emptyset$

entry : [(UNIVERSALSET_COMPLEXSTATE) $\rightarrow$ (UNIVERSALSET_SIMPLESTATE)]

entry_AX1 : AXIOM (entry(c) $\in$ substates(c))

entry_AX2 : AXIOM
($\forall$ (c : (UNIVERSALSET_COMPLEXSTATE)) :
LET entry_state_set : finite_set[StateSDC] = $\{s$: StateSDC $\mid s =$ entry(c)$\}$ IN
(substates(c) \ entry_state_set) $\neq \emptyset$)

hierarchy : [StateSDC $\rightarrow$ finite_set [StateSDC]]

hierarchy_AX1 : AXIOM
($\forall$ (s_1 : StateSDC) :

```
        LET hierarchy_states_set: finite_set[StateSDC]
              = {s3: StateSDC |
                    (∃ (s2: StateSDC): (s3 ∈ hierarchy(s2)) ∧ (s2 ∈ substates(s1)))}
          IN hierarchy(s1) = (singleton(s1) ∪ hierarchy_states_set))
END state_sd
```

D.1.4 Semantic Domain Concept *GRC Class*

```
grcClass_sd : THEORY
 BEGIN

  IMPORTING core_sd, event_sd, state_sd

  GRCClassSDC : TYPE =
  [# porttypes : finite_set [PortTypeSDC] ,
     port_attributes : finite_set [PortTypeAttributeSDC] ,
     data_attributes : finite_set [DataTypeAttributeSDC] ,
     porttype2events_FUN :
        [(porttypes) → finite_set [(UNIVERSALSET_EVENT)]] #]

  g : VAR GRCClassSDC

  p, p1, p2 : VAR PortTypeSDC

  GRCClass_AX1 : AXIOM
    (p ∈ porttypes(g)) ∧ p ≠ NULLPORTTYPE_P0 ⊃
     (∀ (e : (UNIVERSALSET_EVENT)) :
        ((e ∈ porttype2events_FUN(g)(p)) ⊃
          ((e ∈ UNIVERSALSET_INPUTEVENT) ∨ (e ∈ UNIVERSALSET_OUTPUTEVENT)))
         ∧
         LET input_events : finite_set [(UNIVERSALSET_INPUTEVENT)]
                 = {i : (UNIVERSALSET_INPUTEVENT) | (i ∈ porttype2events_FUN(g)(p))},
             output_events : finite_set [(UNIVERSALSET_OUTPUTEVENT)]
                 = {u : (UNIVERSALSET_OUTPUTEVENT) | (u ∈ porttype2events_FUN(g)(p))}
          IN
          (input2external(input_events) ∩ output2external(output_events)) = ∅)

  GRCClass_AX2 : AXIOM
    (p1 ∈ porttypes(g)) ∧ (p2 ∈ porttypes(g)) ∧ p1 ≠ p2 ⊃
     (porttype2events_FUN(g)(p1) ∩ porttype2events_FUN(g)(p2)) = ∅

  GRCClass_AX3 : AXIOM
    (p ∈ porttypes(g)) ∧ p = NULLPORTTYPE_P0 ⊃
     (∀ (e : (UNIVERSALSET_EVENT)) :
        (e ∈ porttype2events_FUN(g)(p)) ⊃
         (e ∈ UNIVERSALSET_INTERNALEVENT))

  GRCClass_AX4 : AXIOM
    ¬ (NULLPORTTYPE_P0 ∈ porttypes(g)) ⊃
     LET event_set : finite_set [EventSDC]
            = {e : EventSDC |
```

```
                (∃ (p: PortTypeSDC):
                   (p ∈ porttypes(g)) ∧ (e ∈ porttype2events_FUN(g)(p)))}
      IN (event_set ∩ UNIVERSALSET_INTERNALEVENT) = ∅

  GRCClass_AX5: AXIOM
    (p ∈ porttypes(g)) ∧ p ≠ NULLPORTTYPE_P0 ⊃
     (∀ (e: (UNIVERSALSET_EVENT)):
        (e ∈ porttype2events_FUN(g)(p)) ⊃
         (e ∈ (UNIVERSALSET_INPUTEVENT ∪ UNIVERSALSET_OUTPUTEVENT)))

  GRCClass_AX6: AXIOM
    LET event_set: finite_set[EventSDC]
            = {e: EventSDC |
                  (p ∈ porttypes(g)) ∧
                   p ≠ NULLPORTTYPE_P0 ∧ (e ∈ porttype2events_FUN(g)(p))}
      IN
      (event_set ⊆ (UNIVERSALSET_INPUTEVENT ∪ UNIVERSALSET_OUTPUTEVENT))

  compatible: [PortTypeSDC, PortTypeSDC → boolean]

  compatible_AX1: AXIOM
    compatible(p1, p2) ⊃
     (∃ (g1, g2: GRCClassSDC):
        g1 ≠ g2 ∧
         (p1 ∈ porttypes(g1)) ∧
          (p2 ∈ porttypes(g2)) ∧
           LET input_events1: finite_set[(UNIVERSALSET_INPUTEVENT)]
                 = {i: (UNIVERSALSET_INPUTEVENT) | (i ∈ porttype2events_FUN(g1)(p1))},
               output_events1: finite_set[(UNIVERSALSET_OUTPUTEVENT)]
                 = {u: (UNIVERSALSET_OUTPUTEVENT) |
                        (u ∈ porttype2events_FUN(g1)(p1))},
               input_events2: finite_set[(UNIVERSALSET_INPUTEVENT)]
                 = {i: (UNIVERSALSET_INPUTEVENT) | (i ∈ porttype2events_FUN(g2)(p2))},
               output_events2: finite_set[(UNIVERSALSET_OUTPUTEVENT)]
                 = {u: (UNIVERSALSET_OUTPUTEVENT) |
                        (u ∈ porttype2events_FUN(g2)(p2))}
             IN
             input2external(input_events1) = output2external(output_events2) ∧
              input2external(input_events2) =
               output2external(output_events1))

  compatible_AX2: AXIOM
    (p1 ∈ porttypes(g)) ∧ (p2 ∈ porttypes(g)) ⊃
     ¬ compatible(p1, p2)
END grcClass_sd
```

D.1.5 Semantic Domain Concept *Extended Statechart*

```
extendedStatechart_sd: THEORY
 BEGIN

  IMPORTING core_sd, event_sd, state_sd

  ClockSDC: TYPE = nat

  GuardSDC: TYPE =
  [# port_condition: [TimeSDC, PortSDC → boolean],
     enabling_condition: [TimeSDC → boolean],
     time_constraint: boolean #]

  ActionSDC: TYPE =
  [# post_condition: [TimeSDC, TimeSDC, PortSDC → boolean],
     clock_init: boolean #]

  TransitionSDC: TYPE =
  [# source: StateSDC,
     destination: StateSDC,
     trigger: EventSDC,
     guard: GuardSDC,
     action: ActionSDC #]

  ExtendedStatechartSDC: TYPE =
  [# states: finite_set[StateSDC],
     clocks: finite_set[ClockSDC],
     transitions: finite_set[TransitionSDC],
     clock_init_FUN: [(clocks) → (transitions)],
     constraint_clocks_FUN: [(transitions) → finite_set[(clocks)]],
     clock_time_FUN: [(clocks), (transitions) → TimeSDC],
     constraint_bounds_FUN: [(clocks), (transitions) → [TimeSDC, TimeSDC]],
     disabling_states_FUN: [(clocks) → finite_set[(states)]] #]

  initial_state_FUN: [ExtendedStatechartSDC → StateSDC]

  s, s1, s2: VAR StateSDC

  r, r1: VAR TransitionSDC

  x: VAR ExtendedStatechartSDC

  State_AX: AXIOM (states(x) ⊆ UNIVERSALSET_STATE)
```

State_AX1 : AXIOM
$(\text{initial\_state\_FUN}(x) \in \text{states}(x)) \;\wedge$
$\quad(\text{initial\_state\_FUN}(x) \in \text{UNIVERSALSET\_SIMPLESTATE})$

State_AX2 : AXIOM
$(s \in \text{states}(x)) \;\supset$
$\quad(s \in \text{UNIVERSALSET\_SIMPLESTATE}) \;\vee$
$\quad\quad(s \in \text{UNIVERSALSET\_COMPLEXSTATE})$

State_AX3 : AXIOM
$(s \in \text{states}(x)) \;\wedge\; (s \in \text{UNIVERSALSET\_COMPLEXSTATE}) \;\supset$
$\quad(\text{entry}(s) \in \text{states}(x)) \;\wedge\; (s \in \text{UNIVERSALSET\_SIMPLESTATE})$

State_AX4 : AXIOM
$(s \in \text{states}(x)) \;\wedge\; (s \in \text{UNIVERSALSET\_COMPLEXSTATE}) \;\supset$
$\quad(\text{substates}(s) \subseteq \text{states}(x))$

State_AX5 : AXIOM
$(s_1 \in \text{states}(x)) \;\wedge$
$\quad(s_1 \in \text{UNIVERSALSET\_COMPLEXSTATE}) \;\wedge$
$\quad\quad(s_2 \in \text{states}(x)) \;\wedge\; (s_2 \in \text{UNIVERSALSET\_COMPLEXSTATE}) \;\wedge\; s_1 \neq s_2$
$\quad\supset\; (\text{substates}(s_1) \cap \text{substates}(s_2)) = \emptyset$

Transition_AX1 : AXIOM
$(r \in \text{transitions}(x)) \;\supset$
$\quad$ LET $\text{all\_substates}: \text{finite\_set}[\text{StateSDC}]$
$\quad\quad\quad = \{s_1 : \text{StateSDC} \mid (s_2 \in \text{states}(x)) \;\wedge\; (s_1 \in \text{substates}(s_2))\}$
$\quad\quad$ IN
$\quad\quad(\text{destination}(r) \in (\text{states}(x) \setminus \text{all\_substates})) \;\vee$
$\quad\quad\quad(\exists\; (s_3 : \text{StateSDC}) :$
$\quad\quad\quad\quad(s_3 \in (\text{states}(x) \setminus \text{all\_substates})) \;\wedge$
$\quad\quad\quad\quad\quad(s_3 \in \text{UNIVERSALSET\_COMPLEXSTATE}) \;\wedge\; \text{destination}(r) = \text{entry}(s_3))$
$\quad\quad\quad\vee$
$\quad\quad\quad(\exists\; (s_4 : \text{StateSDC}) :$
$\quad\quad\quad\quad(s_4 \in \text{states}(x)) \;\wedge$
$\quad\quad\quad\quad\quad(s_4 \in \text{UNIVERSALSET\_COMPLEXSTATE}) \;\wedge$
$\quad\quad\quad\quad\quad\quad(\text{source}(r) \in \text{hierarchy}(s_4)) \;\wedge\; (\text{destination}(r) \in \text{substates}(s_4)))$
$\quad\quad\quad\vee$
$\quad\quad\quad(\exists\; (s_5,\; s_6 : \text{StateSDC}) :$
$\quad\quad\quad\quad(s_5 \in \text{states}(x)) \;\wedge$
$\quad\quad\quad\quad\quad(s_5 \in \text{UNIVERSALSET\_COMPLEXSTATE}) \;\wedge$
$\quad\quad\quad\quad\quad\quad(\text{source}(r) \in \text{hierarchy}(s_5)) \;\wedge$
$\quad\quad\quad\quad\quad\quad\quad(s_6 \in \text{substates}(s_5)) \;\wedge$
$\quad\quad\quad\quad\quad\quad\quad\quad(s_6 \in \text{UNIVERSALSET\_COMPLEXSTATE}) \;\wedge$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\text{destination}(r) = \text{entry}(s_6))$

Transition_AX2 : AXIOM
$(r_1 \in \text{transitions}(x)) \wedge (\text{source}(r_1) \in \text{UNIVERSALSET\_COMPLEXSTATE}) \supset$
LET additional_transitions : finite_set [TransitionSDC]
$= \{r_2 : \text{TransitionSDC} \mid (\text{source}(r_2) \in \text{hierarchy}(\text{source}(r_1)))\}$
IN $(\text{additional\_transitions} \subseteq \text{transitions}(x))$

Transition_AX3 : AXIOM
$(r \in \text{transitions}(x)) \wedge (\text{destination}(r_1) \in \text{UNIVERSALSET\_COMPLEXSTATE}) \supset$
LET additional_transitions : finite_set [TransitionSDC]
$= \{r_2 : \text{TransitionSDC} \mid \text{destination}(r_2) = \text{entry}(\text{destination}(r_1))\}$
IN $(\text{additional\_transitions} \subseteq \text{transitions}(x))$

constraint_clocks_FUN_AX1 : AXIOM
$(r \in \text{transitions}(x)) \wedge (\text{trigger}(r) \in \text{UNIVERSALSET\_INPUTEVENT}) \supset$
$\text{constraint\_clocks\_FUN}(x)(r) = \emptyset$

constraint_clocks_FUN_AX2 : AXIOM
$(r \in \text{transitions}(x)) \wedge \text{constraint\_clocks\_FUN}(x)(r) \neq \emptyset \supset$
$(\text{trigger}(r) \in \text{UNIVERSALSET\_INTERNALEVENT}) \vee$
$(\text{trigger}(r) \in \text{UNIVERSALSET\_OUTPUTEVENT})$

constraint_clocks_FUN_AX3 : AXIOM
$(r_1 \in \text{transitions}(x)) \wedge \text{constraint\_clocks\_FUN}(x)(r_1) \neq \emptyset \supset$
$(\forall\ (c : \text{ClockSDC}) :$
$(c \in \text{constraint\_clocks\_FUN}(x)(r_1)) \supset$
$(\exists\ (r_2 : \text{TransitionSDC}) :$
$(r_2 \in \text{transitions}(x)) \wedge$
$r_2 \neq r_1 \wedge \text{clock\_init\_FUN}(x)(c) = r_2))$

clock_time_FUN_AX1 : AXIOM
$(r_1 \in \text{transitions}(x)) \wedge \text{constraint\_clocks\_FUN}(x)(r_1) \neq \emptyset \supset$
$(\forall\ (c : \text{ClockSDC}) :$
$(c \in \text{constraint\_clocks\_FUN}(x)(r_1)) \wedge$
$(\exists\ (r_2 : \text{TransitionSDC}) :$
$\text{clock\_init\_FUN}(x)(c) = r_2 \supset \text{clock\_time\_FUN}(x)(c, r_2) = 0))$
END extendedStatechart_sd

D.1.6 Semantic Domain Concept *GRC Type*

```
grcType_sd : THEORY
 BEGIN

  IMPORTING grcClass_sd, extendedStatechart_sd

  GRCTypeSDC : TYPE =
  [# grcClass : GRCClassSDC, statechart : ExtendedStatechartSDC #]

  UNIVERSALSET_GRCTYPE : finite_set[GRCTypeSDC]

  ReactiveObjectSDC : TYPE

  instance_of : [ReactiveObjectSDC, GRCTypeSDC → boolean]

  r : VAR TransitionSDC

  y : VAR GRCTypeSDC

  grcType_AX1 : AXIOM
    (r ∈ transitions(statechart(y))) ⊃
     (∃ (pT : PortTypeSDC) :
        (pT ∈ porttypes(grcClass(y))) ∧
         (trigger(r) ∈ porttype2events_FUN(grcClass(y))(pT)))

  grcType_AX2 : AXIOM
    (r ∈ transitions(statechart(y))) ∧ (trigger(r) ∈ UNIVERSALSET_INTERNALEVENT) ⊃
     port_condition(guard(r)) = K_conversion(TRUE)

  grcType_AX3 : AXIOM
    (r ∈ transitions(statechart(y))) ∧ constraint_clocks_FUN(statechart(y))(r) = ∅ ⊃
     time_constraint(guard(r)) = TRUE

  grcType_AX4 : AXIOM
    (r ∈ transitions(statechart(y))) ∧ (trigger(r) ∈ UNIVERSALSET_INPUTEVENT) ⊃
     time_constraint(guard(r)) = TRUE
 END grcType_sd
```

D.1.7 Semantic Domain Concept *Configuration*

```
configuration_sd: THEORY
 BEGIN

  IMPORTING grcType_sd

  ConfigurationSDC: TYPE =
  [# reactive_objects: finite_set[ReactiveObjectSDC],
     port_objects: finite_set[PortSDC],
     port_aggregation_FUN: [(reactive_objects) → finite_set[(port_objects)]],
     port_link_FUN: [(port_objects) → (port_objects)] #]

  p, p1, p2: VAR PortSDC

  y, y1, y2: VAR PortTypeSDC

  r, r1, r2: VAR ReactiveObjectSDC

  f: VAR ConfigurationSDC

  ReactiveObject_AX1: AXIOM
    (r ∈ reactive_objects(f)) ⊃
     (∃ (grcT: GRCTypeSDC):
        (grcT ∈ UNIVERSALSET_GRCTYPE) ∧ instance_of(r, grcT))

  PortObject_AX1: AXIOM
    (p ∈ port_objects(f)) ⊃
     (∃ (y: PortTypeSDC): (y ∈ UNIVERSALSET_PORTTYPE) ∧ (p ∈ y))

  PortOwnership_AX1: AXIOM
    (r1 ∈ reactive_objects(f)) ∧ (r2 ∈ reactive_objects(f)) ∧ r1 ≠ r2 ⊃
     (port_aggregation_FUN(f)(r1) ∩ port_aggregation_FUN(f)(r2)) = ∅

  CommunicationChannel_AX1: AXIOM
    (p1 ∈ port_objects(f)) ∧ (p2 ∈ port_objects(f)) ∧ p1 ≠ p2 ⊃
     port_link_FUN(f)(p1) ≠ port_link_FUN(f)(p2)

  CommunicationChannel_AX2: AXIOM
    (p1 ∈ port_objects(f)) ∧ (p2 ∈ port_objects(f)) ∧ p1 ≠ p2 ∧ port_link_FUN(f)(p1) = p2
     ⊃ port_link_FUN(f)(p2) = p1

  CommunicationChannel_AX3: AXIOM
    (p ∈ port_objects(f)) ∧ (p ∈ y1) ∧ (port_link_FUN(f)(p) ∈ y2) ⊃ compatible(y1, y2)
 END configuration_sd
```

D.1.8 Semantic Domain Concept *Scenario*

```
scenario_sd : THEORY
 BEGIN

  IMPORTING grcType_sd

  MessageSDC : TYPE =
  [# sender : ReactiveObjectSDC,
     receiver : ReactiveObjectSDC,
     event : EventSDC,
     time : TimeSDC #]

  ScenarioSDC : TYPE =
  [# reactive_objects : finite_set[ReactiveObjectSDC],
     messages : finite_sequence[MessageSDC] #]

  r : VAR ReactiveObjectSDC

  n : VAR ScenarioSDC

  m : VAR MessageSDC

  ReactiveObject_AX1 : AXIOM
    (r ∈ reactive_objects(n)) ⊃
     (∃ (g : GRCTypeSDC) :
        (g ∈ UNIVERSALSET_GRCTYPE) ∧ instance_of(r, g))

  Message_AX1 : AXIOM
    (m ∈ finseq2list(messages(n))) ⊃
     (sender(m) ∈ reactive_objects(n))

  Message_AX2 : AXIOM
    (m ∈ finseq2list(messages(n))) ⊃
     (receiver(m) ∈ reactive_objects(n))

  Message_AX3 : AXIOM
    (m ∈ finseq2list(messages(n))) ⊃
     (event(m) ∈ UNIVERSALSET_EVENT)
 END scenario_sd
```

D.1.9 Semantic Domain Concept *Reactive System Model*

```
reactiveSystemModel_sd: THEORY
 BEGIN

  IMPORTING grcType_sd, configuration_sd, scenario_sd

  ReactiveSystemModelSDC: TYPE =
  [# grctypes: finite_set[GRCTypeSDC],
     collaborations: finite_set[ConfigurationSDC],
     scenarios: finite_set[ScenarioSDC] #]

  s: VAR ReactiveSystemModelSDC

  c: VAR ConfigurationSDC

  n: VAR ScenarioSDC

  r: VAR ReactiveObjectSDC

  p: VAR PortSDC

  m: VAR MessageSDC

  ReactiveSystemModel_AX1: AXIOM
    (c ∈ collaborations(s)) ∧ (r ∈ reactive_objects(c)) ⊃
     (∃ (g: GRCTypeSDC): (g ∈ grctypes(s)) ∧ instance_of(r, g))

  ReactiveSystemModel_AX2: AXIOM
    (c ∈ collaborations(s)) ∧ (p ∈ port_objects(c)) ⊃
     (∃ (g: GRCTypeSDC, t: PortTypeSDC):
        (g ∈ grctypes(s)) ∧
         (t ∈ porttypes(grcClass(g))) ∧ (p ∈ t))

  ReactiveSystemModel_AX3: AXIOM
    (n ∈ scenarios(s)) ∧ (r ∈ reactive_objects(n)) ⊃
     (∃ (g: GRCTypeSDC): (g ∈ grctypes(s)) ∧ instance_of(r, g))

  ReactiveSystemModel_AX4: AXIOM
    (n ∈ scenarios(s)) ∧ (m ∈ finseq2list(messages(n))) ⊃
     (∃ (t1, t2: PortTypeSDC, g1, g2: GRCTypeSDC, e1, e2: (UNIVERSALSET_EVENT)):
        (g1 ∈ grctypes(s)) ∧
         (g2 ∈ grctypes(s)) ∧
          (t1 ∈ porttypes(grcClass(g1))) ∧
           (t2 ∈ porttypes(grcClass(g2))) ∧
```

$(e_1 \in$ porttype2events_FUN(grcClass(g_1))(t_1)) $\wedge$
$(e_2 \in$ porttype2events_FUN(grcClass(g_2))(t_2)) $\wedge$
(event(m) $\in$ input2external(singleton(e_1))) $\wedge$
(event(m) $\in$ output2external(singleton(e_2))) $\wedge$
compatible(t_1, t_2))
END reactiveSystemModel_sd

Appendix E

Formalization of Semantic Mapping in PVS

The semantic mapping defines the meaning of each well-formed RTUML model element by specifying the corresponding semantic domain concept. Section 5.6 includes an informal description of the mapping. We provide a PVS specification of the semantic mapping, defining a function for each RTUML model element. The functions are characterized by specifying a set of axioms defining the mapping. The mapping for model elements that take different stereotypes, that is, class, association, classifier role, and association role, is defined by the union of a set of functions, one for each stereotype. Similarly, the mapping for attributes is defined by the union of a set of three functions, one for port type attributes, one for data type attributes, and one for the attribute of a port type. Section E.1 gives the PVS Theory for the semantic mapping.

E.1 PVS Theory for Semantic Mapping

```
semanticMapping_sm: THEORY
 BEGIN

  IMPORTING backbone_abs, relationships_abs, dependencies_abs, classifiers_abs,
            auxiliaryElements_abs, extensionMechanisms_abs, actions_abs, signals_abs,
            collaborations_abs, stateMachines_abs, core_aux, core_sd, event_sd, state_sd,
            grcClass_sd, extendedStatechart_sd, grcType_sd, configuration_sd, scenario_sd,
            reactiveSystemModel_sd

  semanticMapping1SM: [ClassME → GRCClassSDC]

  semanticMapping2SM: [ClassME → PortTypeSDC]

  semanticMapping3SM: [DataTypeME → DataTypeSDC]

  semanticMapping4SM: [BindingME → DataTypeSDC]

  semanticMapping5SM: [AttributeME → PortTypeAttributeSDC]

  semanticMapping6SM: [AttributeME → DataTypeAttributeSDC]

  semanticMapping7SM: [AttributeME → finite_set[(UNIVERSALSET_EVENT)]]

  semanticMapping8SM: [AssociationME → [GRCClassSDC, PortTypeSDC]]

  semanticMapping9SM: [AssociationME → [PortTypeSDC, PortTypeSDC]]

  semanticMapping10SM: [CollaborationME → ConfigurationSDC]

  semanticMapping11SM: [ClassifierRoleME → ReactiveObjectSDC]

  semanticMapping12SM: [ClassifierRoleME → PortSDC]

  semanticMapping13SM: [AssociationRoleME → [ReactiveObjectSDC, PortSDC]]

  semanticMapping14SM: [AssociationRoleME → [PortSDC, PortSDC]]

  semanticMapping15SM: [InteractionME → ScenarioSDC]

  semanticMapping16SM: [MessageME → MessageSDC]

  semanticMapping17SM: [StateMachineME → ExtendedStatechartSDC]
```

semanticMapping18SM: [TransitionME → TransitionSDC]

semanticMapping19SM: [StateME → StateSDC]

semanticMapping20SM: [SimpleStateME → StateSDC]

semanticMapping21SM: [CompositeStateME → StateSDC]

semanticMapping22SM: [EventABS → EventSDC]

semanticMapping23SM: [GuardME → GuardSDC]

semanticMapping24SM: [ActionABS → ActionSDC]

semanticMapping25SM: [ClassME, StateMachineME → GRCTypeSDC]

c: VAR ClassME

a: VAR AssociationME

l: VAR CollaborationME

i: VAR InteractionME

m: VAR StateMachineME

e: VAR CompositeStateME

g: VAR GRCClassSDC

p, p_1, p_2: VAR PortTypeSDC

f: VAR ConfigurationSDC

n: VAR ScenarioSDC

x: VAR ExtendedStatechartSDC

s: VAR StateSDC

y: VAR GRCTypeSDC

semanticMapping1AX1: AXIOM
  semanticMapping1SM(c) = g ⊃
   stereotypeAUX(c) = grcSTE ∧

```
    card(allFeaturesAUX(isClassifier(c))) =
      card(port_attributes(g)) + card(data_attributes(g))

semanticMapping1AX2: AXIOM
  semanticMapping1SM(c) = g ⊃
    LET s: finite_set[StructuralFeatureABS]
            = {f: StructuralFeatureABS |
                  (isaFeature(f) ∈ allFeaturesAUX(isClassifier(c))) ∧
                   (∃ (c2: ClassME):
                      stereotypeAUX(c2) = porttypeSTE ∧ typeASS(f) = isaClassifier(c2))}
      IN card(s) = card(port_attributes(g))

semanticMapping1AX3: AXIOM
  semanticMapping1SM(c) = g ⊃
    LET s: finite_set[StructuralFeatureABS]
            = {f: StructuralFeatureABS |
                  (isaFeature(f) ∈ allFeaturesAUX(isaClassifier(c))) ∧
                   (∃ (d: DataTypeME): typeASS(f) = isaClassifier(d))
                   ∨
                   (∃ (b: BindingME, l: ClassifierABS):
                      (isaModelElement(isaGeneralizableElement(l)) ∈ clientsASS(isaDependency(b)))
                       ∧ typeASS(f) = l)}
      IN card(s) = card(data_attributes(g))

semanticMapping1AX4: AXIOM
  semanticMapping1SM(c) = g ⊃
   (∀ (a: AttributeME):
      (isaFeature(isaStructuralFeature(a)) ∈ allFeaturesAUX(isaClassifier(c))) ⊃
       ((∃ (c2: ClassME, p: PortTypeAttributeSDC):
           typeASS(isaStructuralFeature(a)) = isaClassifier(c2) ∧
            stereotypeAUX(c2) = porttypeSTE ∧
             (p ∈ port_attributes(g)) ∧ semanticMapping5SM(a) = p)
        ∨
        (∃ (d: DataTypeME, d: DataTypeAttributeSDC):
           typeASS(isaStructuralFeature(a)) = isaClassifier(d) ∧
             (d ∈ data_attributes(g)) ∧ semanticMapping6SM(a) = d)
         ∨
         (∃ (b: BindingME, l: ClassifierABS, d: DataTypeAttributeSDC):
            (isaModelElement(isaGeneralizableElement(l)) ∈ clientsASS(isaDependency(b))) ∧
             typeASS(isaStructuralFeature(a)) = l ∧
              (d ∈ data_attributes(g)) ∧
               semanticMapping6SM(a) = d)))

semanticMapping1AX5: AXIOM
  semanticMapping1SM(c) = g ⊃
```

card (associationEndsASS (isaClassifier (c))) = card (porttypes (g))

semanticMapping1AX6 : AXIOM
semanticMapping1SM (c) = g ⊃
(∀ (e : AssociationEndME) :
(e ∈ allOppositeAssociationEndsAUX(isaClassifier(c))) ⊃
(∃ (c_2 : ClassME , p : PortTypeSDC) :
typeASS (e) = isaClassifier (c_2) ∧
stereotypeAUX (c_2) = porttypeSTE ∧
semanticMapping2SM (c_2) = p ∧ (p ∈ porttypes(g))))

semanticMapping2AX1 : AXIOM
semanticMapping2SM (c) = p ⊃
(∀ (e : AssociationEndME) :
(e ∈ allOppositeAssociationEndsAUX(isaClassifier(c))) ⊃
(∃ (c_2 : ClassME) :
typeASS (e) = isaClassifier (c_2) ∧ stereotypeAUX (c_2) = porttypeSTE ⊃
(∃ (p_2 : PortTypeSDC) :
semanticMapping2SM (c_2) = p_2 ∧ compatible (p , p_2))))

semanticMapping2AX2 : AXIOM
semanticMapping2SM (c) = p ⊃
(∀ (e : AssociationEndME) :
(e ∈ allOppositeAssociationEndsAUX(isaClassifier(c))) ⊃
(∃ (c_2 : ClassME) :
typeASS (e) = isaClassifier (c_2) ∧ stereotypeAUX (c_2) = grcSTE ⊃
(∃ (g : GRCClassSDC) :
semanticMapping1SM (c_2) = g ∧ (p ∈ porttypes(g)))))

semanticMapping2AX3 : AXIOM
semanticMapping2SM (c) = p ⊃
(∃ (e : AssociationEndME , c_2 : ClassME) :
(e ∈ allOppositeAssociationEndsAUX(isaClassifier(c))) ∧
typeASS (e) = isaClassifier (c_2) ∧
stereotypeAUX (c_2) = grcSTE ∧
(∃ (g : GRCClassSDC) :
semanticMapping1SM (c_2) = g ∧
(p ∈ porttypes(g)) ∧
(∃ (a : AttributeME) :
(a ∈ allAttributesAUX(isaClassifier(c))) ∧
semanticMapping7SM (a) = porttype2events_FUN (g) (p))))

semanticMapping8AX1 : AXIOM
semanticMapping8SM (a) = (g , p) ⊃
stereotypeAUX (a) = portaggregationSTE ∧ (p ∈ porttypes(g))

```
semanticMapping9AX1 : AXIOM
  semanticMapping9SM(a) = (p1, p2) ⊃
   stereotypeAUX(a) = portlinkSTE ∧ compatible(p1, p2)

semanticMapping10AX1 : AXIOM
  semanticMapping10SM(l) = f ⊃
   card(ownedElements(isaNameSpace(l))) =
    card(reactive_objects(f)) + card(port_objects(f))

semanticMapping10AX2 : AXIOM
  semanticMapping10SM(l) = f ⊃
   (∀ (r: ClassifierRoleME):
      (isaModelElement(isaGeneralizableElement(isaClassifier(r))) ∈ ownedElements(isaNameSpace(l)))
       ⊃
       ((∃ (c: ClassME):
            stereotypeAUX(c) = grcSTE ∧
             baseASS(r) = isaClassifier(c) ∧ (semanticMapping11SM(r) ∈ reactive_objects(f)))
         ∨
         (∃ (c: ClassME):
            stereotypeAUX(c) = porttypeSTE ∧
             baseASS(r) = isaClassifier(c) ∧
              (semanticMapping12SM(r) ∈ port_objects(f)))))

semanticMapping10AX3 : AXIOM
  semanticMapping10SM(l) = f ⊃
   (∀ (a: AssociationRoleME):
      (isaModelElement(isaRelationship(isaAssociation(a))) ∈ ownedElements(isaNameSpace(l)))
       ⊃
       ((∃ (r: ReactiveObjectSDC, p: PortSDC):
            stereotypeAUX(baseASS(a)) = portaggregationSTE ∧
             semanticMapping13SM(a) = (r, p) ∧
              (r ∈ reactive_objects(f)) ∧
               (p ∈ port_objects(f)) ∧ (p ∈ port_aggregation_FUN(f)(r)))
         ∨
         (∃ (p1, p2: PortSDC):
            stereotypeAUX(baseASS(a)) = portlinkSTE ∧
             semanticMapping14SM(a) = (p1, p2) ∧
              (p1 ∈ port_objects(f)) ∧
               (p2 ∈ port_objects(f)) ∧ port_link_FUN(f)(p1) = p2)))

semanticMapping15AX1 : AXIOM
  semanticMapping15SM(i) = n ⊃ card(messages(i)) = length(messages(n))

semanticMapping15AX2 : AXIOM
```

$$
\begin{array}{l}
\quad \text{semanticMapping15SM}(i) \ = \ n \ \supset \\
\quad\quad (\forall \ (m: \ \text{MessageME}): \\
\quad\quad\quad (m \in \text{messages}(i)) \ \supset \\
\quad\quad\quad\quad (\text{semanticMapping16SM}(m) \in \text{finseq2list}(\text{messages}(n))))
\end{array}
$$

$$
\begin{array}{l}
\text{semanticMapping15AX3}: \ \text{AXIOM} \\
\quad \text{semanticMapping15SM}(i) \ = \ n \ \supset \\
\quad\quad (\forall \ (m_1: \ \text{MessageME}): \\
\quad\quad\quad (m_1 \in \text{messages}(i)) \ \wedge \\
\quad\quad\quad\quad (\exists \ (m_2: \ \text{MessageSDC}): \\
\quad\quad\quad\quad\quad (m_2 \in \text{finseq2list}(\text{messages}(n))) \ \wedge \\
\quad\quad\quad\quad\quad\quad \text{semanticMapping16SM}(m_1) \ = \ m_2 \ \wedge \\
\quad\quad\quad\quad\quad\quad\quad \text{semanticMapping11SM}(\text{senderASS}(m_1)) \ = \ \text{sender}(m_2) \ \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad \text{semanticMapping11SM}(\text{receiverASS}(m_1)) \ = \ \text{receiver}(m_2)))
\end{array}
$$

$$
\begin{array}{l}
\text{semanticMapping17AX1}: \ \text{AXIOM} \\
\quad \text{semanticMapping17SM}(m) \ = \ x \ \supset \\
\quad\quad \text{semanticMapping19SM}(\text{top}(m)) \ = \ \text{initial\_state\_FUN}(x)
\end{array}
$$

$$
\begin{array}{l}
\text{semanticMapping17AX2}: \ \text{AXIOM} \\
\quad \text{semanticMapping17SM}(m) \ = \ x \ \supset \\
\quad\quad \text{card}(\text{transitions}(m)) \ = \ \text{card}(\text{transitions}(x))
\end{array}
$$

$$
\begin{array}{l}
\text{semanticMapping17AX3}: \ \text{AXIOM} \\
\quad \text{semanticMapping17SM}(m) \ = \ x \ \supset \\
\quad\quad (\forall \ (t_1: \ \text{TransitionME}): \\
\quad\quad\quad (t_1 \in \text{transitions}(m)) \ \supset \\
\quad\quad\quad\quad (\exists \ (t_2: \ \text{TransitionSDC}): \\
\quad\quad\quad\quad\quad (t_2 \in \text{transitions}(x)) \ \wedge \ \text{semanticMapping18SM}(t_1) \ = \ t_2))
\end{array}
$$

$$
\begin{array}{l}
\text{semanticMapping17AX4}: \ \text{AXIOM} \\
\quad \text{semanticMapping17SM}(m) \ = \ x \ \supset \\
\quad\quad (\forall \ (t_1: \ \text{TransitionME}): \\
\quad\quad\quad (t_1 \in \text{transitions}(m)) \ \supset \\
\quad\quad\quad\quad (\exists \ (t_2: \ \text{TransitionSDC}, \ s_1, \ s_2: \ \text{StateME}): \\
\quad\quad\quad\quad\quad (t_2 \in \text{transitions}(x)) \ \wedge \\
\quad\quad\quad\quad\quad\quad \text{semanticMapping18SM}(t_1) \ = \ t_2 \ \wedge \\
\quad\quad\quad\quad\quad\quad\quad \text{sourceASS}(t_1) \ = \ \text{isaStateVertex}(s_1) \ \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad \text{semanticMapping19SM}(s_1) \ = \ \text{source}(t_2) \ \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{targetASS}(t_1) \ = \ \text{isaStateVertex}(s_2) \ \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{semanticMapping19SM}(s_2) \ = \ \text{destination}(t_2)))
\end{array}
$$

$$
\begin{array}{l}
\text{semanticMapping17AX5}: \ \text{AXIOM} \\
\quad \text{semanticMapping17SM}(m) \ = \ x \ \supset \\
\quad\quad (\forall \ (t_1: \ \text{TransitionME}):
\end{array}
$$

$(t_1 \in \text{transitions}(m)) \supset$
$(\exists\ (t_2 : \text{TransitionSDC}) :$
$(t_2 \in \text{transitions}(x)) \wedge$
$\text{semanticMapping18SM}(t_1) = t_2 \wedge$
$(\exists\ (s : \text{SimpleStateME}) :$
$\text{sourceASS}(t_1) = \text{isaStateVertex}(\text{isaState}(s)) \supset$
$\text{semanticMapping20SM}(s) = \text{source}(t_2) \wedge \text{substates}(\text{source}(t_2)) = \emptyset)$
$\wedge$
$(\exists\ (s : \text{SimpleStateME}) :$
$\text{targetASS}(t_1) = \text{isaStateVertex}(\text{isaState}(s)) \supset$
$\text{semanticMapping20SM}(s) = \text{destination}(t_2) \wedge$
$\text{substates}(\text{destination}(t_2)) = \emptyset)))$

semanticMapping17AX6 : AXIOM
$\text{semanticMapping17SM}(m) = x \supset$
$(\forall\ (t_1 : \text{TransitionME}) :$
$(t_1 \in \text{transitions}(m)) \supset$
$(\exists\ (t_2 : \text{TransitionSDC}) :$
$(t_2 \in \text{transitions}(x)) \wedge$
$\text{semanticMapping18SM}(t_1) = t_2 \wedge$
$(\exists\ (\text{cs} : \text{CompositeStateME}) :$
$\text{sourceASS}(t_1) = \text{isaStateVertex}(\text{isaState}(\text{cs})) \supset$
$\text{semanticMapping21SM}(\text{cs}) = \text{source}(t_2) \wedge \text{substates}(\text{source}(t_2)) \neq \emptyset)$
$\wedge$
$(\exists\ (\text{cs} : \text{CompositeStateME}) :$
$\text{targetASS}(t_1) = \text{isaStateVertex}(\text{isaState}(\text{cs})) \supset$
$\text{semanticMapping21SM}(\text{cs}) = \text{destination}(t_2) \wedge$
$\text{substates}(\text{destination}(t_2)) \neq \emptyset)))$

semanticMapping17AX7 : AXIOM
$\text{semanticMapping17SM}(m) = x \supset$
$(\forall\ (t_1 : \text{TransitionME}) :$
$(t_1 \in \text{transitions}(m)) \supset$
$(\exists\ (t_2 : \text{TransitionSDC},\ e_1 : \text{EventABS}) :$
$(t_2 \in \text{transitions}(x)) \wedge$
$\text{semanticMapping18SM}(t_1) = t_2 \wedge$
$(e_1 \in \text{trigger}(t_1)) \wedge$
$\text{semanticMapping22SM}(e_1) = \text{trigger}(t_2)))$

semanticMapping17AX8 : AXIOM
$\text{semanticMapping17SM}(m) = x \supset$
$(\forall\ (t_1 : \text{TransitionME}) :$
$(t_1 \in \text{transitions}(m)) \supset$
$(\exists\ (t_2 : \text{TransitionSDC},\ g : \text{GuardME}) :$
$(t_2 \in \text{transitions}(x)) \wedge$

semanticMapping18SM(t_1) = t_2 ∧
($g \in$ guard(t_1)) ∧ semanticMapping23SM(g) = guard(t_2)))

semanticMapping17AX9: AXIOM
semanticMapping17SM(m) = x ⊃
(∀ (t_1: TransitionME) :
($t_1 \in$ transitions(m)) ⊃
(∃ (t_2: TransitionSDC, a: ActionABS) :
($t_2 \in$ transitions(x)) ∧
semanticMapping18SM(t_1) = t_2 ∧
($a \in$ effect(t_1)) ∧ semanticMapping24SM(a) = action(t_2)))

semanticMapping21AX1: AXIOM
semanticMapping21SM(e) = s ⊃
card(subvertices(e)) = card(substates(s))

semanticMapping21AX2: AXIOM
semanticMapping21SM(e) = s ⊃
(∀ (s_2: StateME) :
(isaStateVertex(s_2) $\in$ subvertices(e)) ⊃
(semanticMapping19SM(s_2) $\in$ substates(s)))

semanticMapping25AX1: AXIOM
semanticMapping25SM(c, m) = y ⊃
stereotypeAUX(c) = grcSTE ∧
semanticMapping1SM(c) = grcClass(y) ∧
semanticMapping17SM(m) = statechart(y)
END semanticMapping_sm

Appendix F

Formalization of Operational Semantics in PVS

The RTUML operational semantics specified in OCL and given in Chapter 5 correspond to an axiomatization of the abstract reactive object model. This semantics includes a behavioral description for instances of generic reactive classes and subsystems, providing a foundation for the mechanized verification methodology described in Chapter 6. To synthesize this basis with the methodology, we provide a PVS specification of the operational semantics, making use of the type definitions for the semantic domain concepts. Section F.1 gives the PVS Theory for RTUML operational semantics.

F.1 PVS Theory for RTUML Operational Semantics

operationalSemantics_os : THEORY
BEGIN

IMPORTING core_sd, event_sd, state_sd, grcClass_sd, extendedStatechart_sd, grcType_sd, configuration_sd, scenario_sd, reactiveSystemModel_sd

TimeIntervalSDC : TYPE = [# lower: TimeSDC, upper: TimeSDC #]

BeforeTP(T_1, T_2: TimeIntervalSDC) : boolean = upper(T_1) $<$ lower(T_2)

MeetTP(T_1, T_2: TimeIntervalSDC) : boolean = upper(T_1) $=$ lower(T_2)

OverlapsTP(T_1, T_2: TimeIntervalSDC) : boolean =
lower(T_1) $<$ lower(T_2) $\wedge$ lower(T_2) $<$ upper(T_1) $\wedge$ upper(T_1) $<$ upper(T_2)

EqualTP(T_1, T_2: TimeIntervalSDC) : boolean =
lower(T_1) $=$ lower(T_2) $\wedge$ upper(T_1) $=$ upper(T_2)

DuringTP(T_1, T_2: TimeIntervalSDC) : boolean =
lower(T_2) $<$ lower(T_1) $\wedge$ upper(T_1) $<$ upper(T_2)

StartsTP(T_1, T_2: TimeIntervalSDC) : boolean =
lower(T_1) $=$ lower(T_2) $\wedge$ upper(T_1) $<$ upper(T_2)

FinishesTP(T_1, T_2: TimeIntervalSDC) : boolean =
lower(T_1) $<$ lower(T_2) $\wedge$ upper(T_1) $=$ upper(T_2)

WithinTP(t_1, l, u, t_2: TimeSDC) : boolean =
$t_1 + l \leq t_2 \wedge t_2 \leq t_2 + u$

HoldAt : [ReactiveObjectSDC $\rightarrow$ [StateSDC, TimeSDC $\rightarrow$ boolean]]

HoldDuring : [ReactiveObjectSDC $\rightarrow$ [StateSDC, TimeIntervalSDC $\rightarrow$ boolean]]

Occur : [ReactiveObjectSDC $\rightarrow$ [EventSDC, PortSDC, TimeSDC $\rightarrow$ boolean]]

Trigger : [ReactiveObjectSDC $\rightarrow$ [EventSDC, TimeSDC $\rightarrow$ boolean]]

Disable : [ReactiveObjectSDC $\rightarrow$ [EventSDC, TimeSDC $\rightarrow$ boolean]]

Enable : [ReactiveObjectSDC $\rightarrow$ [EventSDC, TimeSDC, TimeSDC $\rightarrow$ boolean]]

b : VAR ReactiveObjectSDC

g: VAR GRCTypeSDC

s, s_1, s_2: VAR StateSDC

t, t_1, t_2, t_3: VAR TimeSDC

T, T_1, T_2: VAR TimeIntervalSDC

e, e_1, e_2: VAR (UNIVERSALSET_EVENT)

p, p_1, p_2, pObj1, pObj2: PortSDC

f: VAR ConfigurationSDC

TimeInterval_AX: AXIOM lower(T) $<$ upper(T)

HoldDuring_AX: AXIOM
  HoldDuring(b)(s, T) $\supset$ lower(T) $\leq t \wedge t \leq$ upper(T) $\supset$ HoldAt(b)(s, t)

Trigger_AX: AXIOM
  instance_of(b, g) $\wedge$ Trigger(b)(e, t) $\supset$
   ($\exists$ (r_1, r_2: TransitionSDC, c: ClockSDC):
     $r_2 \neq r_1 \wedge$
      ($r_1 \in$ transitions(statechart(g))) $\wedge$
       ($r_2 \in$ transitions(statechart(g))) $\wedge$
        ($c \in$ constraint_clocks_FUN(statechart(g))(r_1)) $\wedge$
         clock_init_FUN(statechart(g))(c) $= r_2 \wedge$
          Occur(b)(trigger(r_1), p, t) $\wedge$
           HoldAt(b)(source(r_1), t_1) $\wedge$
            HoldAt(b)(destination(r_1), t_2) $\wedge$
             $t_1 < t \wedge t < t_2 \wedge$ trigger(r_2) $= e$)

Disable_AX: AXIOM
  instance_of(b, g) $\wedge$ Disable(b)(e, t) $\supset$
   ($\exists$ (r_1, r_2: TransitionSDC, c: ClockSDC, s: StateSDC):
     $r_2 \neq r_1 \wedge$
      ($r_1 \in$ transitions(statechart(g))) $\wedge$
       ($r_2 \in$ transitions(statechart(g))) $\wedge$
        ($c \in$ constraint_clocks_FUN(statechart(g))(r_1)) $\wedge$
         clock_init_FUN(statechart(g))(c) $= r_2 \wedge$
          trigger(r_2) $= e \wedge$
           $t <$ (clock_time_FUN(statechart(g))(c, r_2) $+$
                PROJ_2(constraint_bounds_FUN(statechart(g))(c, r_1))) $\wedge$
            ($s \in$ disabling_states_FUN(statechart(g))(c)) $\wedge$ HoldAt(b)(s, t))

OS_AtomicEvent_AX1 : AXIOM
instance_of$(b,\ g) \supset$
$(\forall\ (r_1,\ r_2\text{: TransitionSDC}):$
$(r_1 \in \text{transitions(statechart}(g))) \ \wedge$
$(r_2 \in \text{transitions(statechart}(g))) \ \wedge$
$r_1 \neq r_2 \ \wedge \ \text{trigger}(r_1) = e_1 \ \wedge \ \text{trigger}(r_2) = e_2 \ \wedge \ e_1 \neq e_2 \ \wedge \ \text{Occur}(b)(e_1,\ p_1,\ t)$
$\supset \neg \ \text{Occur}(b)(e_2,\ p_2,\ t))$

OS_AtomicEvent_AX2 : AXIOM
instance_of$(b,\ g) \supset$
$(\forall\ (r\text{: TransitionSDC}):$
$(r \in \text{transitions(statechart}(g))) \ \wedge$
$\text{trigger}(r) = e \ \wedge \ \text{Occur}(b)(e,\ p_1,\ t) \ \wedge \ \text{Occur}(b)(e,\ p_2,\ t)$
$\supset p_1 = p_2)$

OS_StateHierarchy_AX1 : AXIOM
instance_of$(b,\ g) \supset$
$(\forall\ (r\text{: TransitionSDC},\ s_1\text{: StateSDC}):$
$(r \in \text{transitions(statechart}(g))) \ \wedge$
$(\text{source}(r) = s_1 \ \vee \ \text{destination}(r) = s_1) \ \wedge$
$(s_1 \in \text{UNIVERSALSET\_COMPLEXSTATE}) \ \wedge$
$(\exists\ (s_2\text{: StateSDC}):$
$(s_2 \in \text{substates}(s_1)) \ \wedge \ \text{HoldDuring}(b)(s_2,\ T) \supset$
$\text{HoldDuring}(b)(s_1,\ T)))$

OS_StateHierarchy_AX2 : AXIOM
instance_of$(b,\ g) \supset$
$(\forall\ (r\text{: TransitionSDC},\ s_1\text{: StateSDC}):$
$(r \in \text{transitions(statechart}(g))) \ \wedge$
$(\text{source}(r) = s_1 \ \vee \ \text{destination}(r) = s_1) \ \wedge$
$(s_1 \in \text{UNIVERSALSET\_COMPLEXSTATE}) \ \wedge \ \text{HoldDuring}(b)(s_1,\ T)$
$\supset$
$(\exists\ (s_2\text{: StateSDC}):$
$(s_2 \in \text{substates}(s_1)) \ \wedge \ \text{HoldDuring}(b)(s_2,\ T)))$

OS_StateUniqueness_AX : AXIOM
$\text{HoldAt}(b)(s_1,\ t) \ \wedge$
$(\forall\ (s_2\text{: StateSDC}):$
$s_2 \neq s_1 \ \wedge \ \neg\ (s_1 \in \text{substates}(s_2)) \ \wedge \ \neg\ (s_2 \in \text{substates}(s_1)) \supset$
$\neg\ \text{HoldAt}(b)(s_2,\ t))$

OS_Occurrence_AX : AXIOM
$\text{Occur}(b)(e,\ p,\ t) \supset$
$(\exists\ (g\text{: GRCTypeSDC},\ r\text{: TransitionSDC},\ s\text{: StateSDC}):$

```
        instance_of(b, g) ∧
         (r ∈ transitions(statechart(g))) ∧
          source(r) = s ∧
           HoldAt(b)(s, t) ∧
            enabling_condition(guard(r))(t) = TRUE ∧
             port_condition(guard(r))(t, p) = TRUE)

OS_Transition_AX: AXIOM
  HoldAt(b)(s1, t1) ∧ Occur(b)(e, p, t1) ∧ t1 < t2 ⊃
   (∃ (g: GRCTypeSDC, r: TransitionSDC):
        instance_of(b, g) ∧
         (r ∈ transitions(statechart(g))) ∧
          source(r) = s1 ∧
           destination(r) = s2 ∧
            HoldAt(b)(s2, t2) ∧
             post_condition(action(r))(t1, t2, p) = TRUE)

OS_Persistence_AX: AXIOM
  HoldDuring(b)(s, T1) ∧
   MeetTP(T1, T2) ∧
    (∀ (r: TransitionSDC):
        (r ∈ transitions(statechart(g))) ∧ (source(r) = s ⊃ ¬ Occur(b)(trigger(r), p, t)) ⊃
         HoldDuring(b)(s, T2))

OS_Activation_AX: AXIOM
  instance_of(b, g) ⊃
   (∀ (r: TransitionSDC, e: EventSDC):
       (r ∈ transitions(statechart(g))) ∧
        constraint_clocks_FUN(statechart(g))(r) ≠ ∅ ∧
         trigger(r) = e ∧ Trigger(b)(e, t1) ∧ ¬ Disable(b)(e, t2) ∧ t1 < t2
        ⊃ Enable(b)(e, t1, t2))

OS_ConstrainedEvent_AX: AXIOM
  instance_of(b, g) ⊃
   (∀ (r1: TransitionSDC):
       (r1 ∈ transitions(statechart(g))) ∧
        constraint_clocks_FUN(statechart(g))(r1) ≠ ∅ ∧ Occur(b)(trigger(r1), p1, t1)
        ⊃
        (∃ (r2: TransitionSDC, c: ClockSDC):
            r2 ≠ r1 ∧ (r2 ∈ transitions(statechart(g))) ∧
             (c ∈ constraint_clocks_FUN(statechart(g))(r1)) ∧
              clock_init_FUN(statechart(g))(c) = r2 ∧
               Occur(b)(trigger(r2), p2, t2) ∧
                t2 + PROJ_1(constraint_bounds_FUN(statechart(g))(c, r1)) ≤ t1 ∧
                 t1 ≤ t2 + PROJ_2(constraint_bounds_FUN(statechart(g))(c, r1))))
```

OS_Enabling_AX: AXIOM
instance_of(b, g) $\supset$
($\forall$ (r: TransitionSDC, e: EventSDC):
($r \in$ transitions(statechart(g))) $\wedge$
constraint_clocks_FUN(statechart(g))(r) $\neq \emptyset$ $\wedge$
trigger(r) $= e$ $\wedge$
Enable(b)(e, t_1, t_2) $\wedge \neg$ Occur(b)(e, p, t_2) $\wedge$ $t_2 < t_3$ $\wedge \neg$ Disable(b)(e, t_3)
$\supset$ Enable(b)(e, t_1, t_3))

OS_Disabling_AX: AXIOM
instance_of(b, g) $\supset$
($\forall$ (r: TransitionSDC, e: EventSDC):
($r \in$ transitions(statechart(g))) $\wedge$
constraint_clocks_FUN(statechart(g))(r) $\neq \emptyset$ $\wedge$
trigger(r) $= e$ $\wedge$ Enable(b)(e, t_1, t_2) $\wedge$ $t_2 < t_3$ $\wedge$ Disable(b)(e, t_3)
$\supset \neg$ Enable(b)(e, t_1, t_3))

OS_Firing_AX: AXIOM
instance_of(b, g) $\supset$
($\forall$ (r: TransitionSDC, e: EventSDC):
($r \in$ transitions(statechart(g))) $\wedge$
($\exists$ (c: ClockSDC):
($c \in$ constraint_clocks_FUN(statechart(g))(r)) $\wedge$
trigger(r) $= e$ $\wedge$
Enable(b)(e, t_1, t_2) $\wedge$
Occur(b)(e, p, t_2) $\wedge$
WithinTP(t_1, 0, PROJ_2(constraint_bounds_FUN(statechart(g))(c, r)), t_2) $\wedge$
$t_2 < t_3$
$\supset \neg$ Enable(b)(e, t_1, t_3)))

OS_Prohibition_AX: AXIOM
instance_of(b, g) $\supset$
($\forall$ (r: TransitionSDC, e: EventSDC):
($r \in$ transitions(statechart(g))) $\wedge$
($\exists$ (c: ClockSDC):
($c \in$ constraint_clocks_FUN(statechart(g))(r)) $\wedge$
trigger(r) $= e$ $\wedge$
Enable(b)(e, t_1, t_2) $\wedge$
WithinTP(t_1, 0, PROJ_1(constraint_bounds_FUN(statechart(g))(c, r)), t_2)
$\supset \neg$ Occur(b)(e, p, t_2)))

OS_Obligation_AX: AXIOM
instance_of(b, g) $\supset$
($\forall$ (r: TransitionSDC, e: EventSDC):

```
        (r ∈ transitions(statechart(g))) ∧
         (∃ (c: ClockSDC):
            (c ∈ constraint_clocks_FUN(statechart(g))(r)) ∧
             trigger(r) = e ∧
              Enable(b)(e, t1, t2) ∧
               (WithinTP(t1, 0, PROJ_2(constraint_bounds_FUN(statechart(g))(c, r)), t3) ⊃
                 ¬ Disable(b)(e, t3))
             ⊃
             Occur(b)(e, p, t3) ∧
              WithinTP(t1, PROJ_1(constraint_bounds_FUN(statechart(g))(c, r)),
                         PROJ_2(constraint_bounds_FUN(statechart(g))(c, r)),
                         t3)))

  OS_Validity_AX: AXIOM
    instance_of(b, g) ⊃
     (∀ (r: TransitionSDC, e: EventSDC):
        (r ∈ transitions(statechart(g))) ∧
         (∃ (c: ClockSDC):
            (c ∈ constraint_clocks_FUN(statechart(g))(r)) ∧
             trigger(r) = e ∧ Enable(b)(e, t1, t2)
             ⊃
             Trigger(b)(e, t1) ∧
              WithinTP(t1, 0, PROJ_2(constraint_bounds_FUN(statechart(g))(c, r)),
                         t2)))

  OS_Synchrony_AX: AXIOM
    (p1 ∈ port_objects(f)) ∧
     (p2 ∈ port_objects(f)) ∧ port_link_FUN(f)(p1) = p2
     ⊃
     (∃ (b1, b2: ReactiveObjectSDC):
        b1 ≠ b2 ∧
         (p1 ∈ port_aggregation_FUN(f)(b1)) ∧
          (p2 ∈ port_aggregation_FUN(f)(b2)) ∧
           (∀ (e1, e2: (UNIVERSALSET_EVENT)):
              (input2external(singleton(e1)) = output2external(singleton(e2)) ∨
                input2external(singleton(e2)) = output2external(singleton(e1)))
               ⊃
               ((Occur(b1)(e1, p1, t1) ⊃ Occur(b2)(e2, p2, t1)) ∧
                 (Occur(b2)(e1, p2, t2) ⊃ Occur(b1)(e2, p1, t2)))))
END operationalSemantics_os
```

Appendix G

PVS Specifications for Railroad Crossing System

The generalized railroad crossing problem, used in the case study for the verification methodology, is considered as a benchmark problem for evaluating formal methods for specifying, designing, and analyzing real-time systems, and to assess the suitability and scalability of the methods for developing practical systems. In this appendix, we give an axiomatic description of the system in the specification language of PVS, and include the PVS theorem specifying the safety property, and the proof of the property. Section G.1 contains the PVS Theories for the specification of the computational model, the transition time function, the generic reactive classes, the subsystem, and the safety property. Section G.2 contains the sequence of proof commands for the safety property.

G.1 PVS Theories for Railroad Crossing System

G.1.1 PVS Theories *model* and *transition_time*

```
model: THEORY
 BEGIN

  Time: TYPE = {r: real | r ≥ 0}

  Period: TYPE = posnat

  Occurrence: TYPE = posnat
 END model

transition_time[GRC: TYPE, GRC_Event: TYPE]: THEORY
 BEGIN

  IMPORTING model

  TT: [GRC → [Period → [GRC_Event → [Occurrence → Time]]]]
 END transition_time
```

G.1.2 PVS Theory *train*

```
train: THEORY
 BEGIN

  Train_GRC: TYPE+

  Train_Event: TYPE = {e_Near, e_In, e_Out, e_Exit}

  IMPORTING transition_time[Train_GRC, Train_Event]

  i: VAR Period

  j: VAR Occurrence

  r: VAR Train_GRC

  TR_AX_1: AXIOM TT(r)(i)(e_Near)(1) < TT(r)(i)(e_In)(1)

  TR_AX_2: AXIOM TT(r)(i)(e_In)(1) < TT(r)(i)(e_Out)(1)

  TR_AX_3: AXIOM TT(r)(i)(e_Out)(1) < TT(r)(i)(e_Exit)(1)

  TC_AX_1: AXIOM
    TT(r)(i)(e_In)(1) > TT(r)(i)(e_Near)(1) + 2 ∧
     TT(r)(i)(e_In)(1) < TT(r)(i)(e_Near)(1) + 4

  TC_AX_2: AXIOM
    TT(r)(i)(e_Exit)(1) > TT(r)(i)(e_Near)(1) ∧
     TT(r)(i)(e_Exit)(1) < TT(r)(i)(e_Near)(1) + 6
 END train
```

G.1.3 PVS Theory *controller*

```
controller: THEORY
 BEGIN

  Controller_GRC: TYPE+

  Controller_Event: TYPE = {e_Near, e_Exit, e_Lower, e_Raise}

  IMPORTING transition_time[Controller_GRC, Controller_Event]

  i: VAR Period

  j: VAR Occurrence

  c: VAR Controller_GRC

  FirstNear, LastExit: Occurrence

  TR_AX_4: AXIOM TT(c)(i)(e_Near)(FirstNear) < TT(c)(i)(e_Lower)(1)

  TR_AX_5: AXIOM TT(c)(i)(e_Lower)(1) < TT(c)(i)(e_Exit)(j)

  TR_AX_6: AXIOM TT(c)(i)(e_Exit)(j) < TT(c)(i)(e_Raise)(1)

  TC_AX_3: AXIOM
    TT(c)(i)(e_Lower)(1) > TT(c)(i)(e_Near)(FirstNear) ∧
     TT(c)(i)(e_Lower)(1) < TT(c)(i)(e_Near)(FirstNear) + 1

  TC_AX_4: AXIOM
    TT(c)(i)(e_Raise)(1) > TT(c)(i)(e_Exit)(LastExit) ∧
     TT(c)(i)(e_Raise)(1) < TT(c)(i)(e_Exit)(LastExit) + 1

  FirstNear_TR_AX: AXIOM
    (j ≠ FirstNear) ⊃ TT(c)(i)(e_Near)(j) > TT(c)(i)(e_Near)(FirstNear)

  LastExit_TR_AX: AXIOM
    (j ≠ LastExit) ⊃ TT(c)(i)(e_Exit)(j) < TT(c)(i)(e_Exit)(LastExit)
 END controller
```

G.1.4 PVS Theory *gate*

```
gate: THEORY
 BEGIN

  Gate_GRC: TYPE+

  Gate_Event: TYPE = {e_Lower, e_Raise, e_Down, e_Up}

  IMPORTING transition_time[Gate_GRC, Gate_Event]

  i: VAR Period

  j: VAR Occurrence

  g: VAR Gate_GRC

  TR_AX_7: AXIOM TT(g)(i)(e_Lower)(1) < TT(g)(i)(e_Down)(1)

  TR_AX_8: AXIOM TT(g)(i)(e_Down)(1) < TT(g)(i)(e_Raise)(1)

  TR_AX_9: AXIOM TT(g)(i)(e_Raise)(1) < TT(g)(i)(e_Up)(1)

  TC_AX_5: AXIOM
    TT(g)(i)(e_Down)(1) > TT(g)(i)(e_Lower)(1) ∧
     TT(g)(i)(e_Down)(1) < TT(g)(i)(e_Lower)(1) + 1

  TC_AX_6: AXIOM
    TT(g)(i)(e_Up)(1) > TT(g)(i)(e_Raise)(1) + 1 ∧
     TT(g)(i)(e_Up)(1) < TT(g)(i)(e_Raise)(1) + 2
 END gate
```

G.1.5 PVS Theory *railroad*

```
railroad : THEORY
 BEGIN

  IMPORTING train, controller, gate

  i: VAR Period

  j: VAR Occurrence

  r: VAR Train_GRC

  C: Controller_GRC

  G: Gate_GRC

  FirstTrain, LastTrain: Train_GRC

  SY_AX_1: AXIOM
    (∃ j: TT(C)(i)(e_Near)(j) = TT(r)(i)(e_Near)(1)) ∧
     (∃ r: TT(C)(i)(e_Near)(j) = TT(r)(i)(e_Near)(1))

  SY_AX_2: AXIOM
    (∃ j: TT(C)(i)(e_Exit)(j) = TT(r)(i)(e_Exit)(1)) ∧
     (∃ r: TT(C)(i)(e_Exit)(j) = TT(r)(i)(e_Exit)(1))

  SY_AX_3: AXIOM TT(C)(i)(e_Lower)(1) = TT(G)(i)(e_Lower)(1)

  SY_AX_4: AXIOM TT(C)(i)(e_Raise)(1) = TT(G)(i)(e_Raise)(1)

  FirstTrain_TR_AX: AXIOM
    (r ≠ FirstTrain) ⊃ TT(r)(i)(e_Near)(1) > TT(FirstTrain)(i)(e_Near)(1)

  LastTrain_TR_AX: AXIOM
    (r ≠ LastTrain) ⊃ TT(r)(i)(e_Exit)(1) < TT(LastTrain)(i)(e_Exit)(1)

  FirstNear_SY_AX: AXIOM
    TT(C)(i)(e_Near)(FirstNear) = TT(FirstTrain)(i)(e_Near)(1)

  LastExit_SY_AX: AXIOM
    TT(C)(i)(e_Exit)(LastExit) = TT(LastTrain)(i)(e_Exit)(1)
 END railroad
```

G.1.6 PVS Theory *railroad_safety*

```
railroad_safety : THEORY
 BEGIN

  IMPORTING railroad

  i : VAR Period

  r : VAR Train_GRC

  safety : THEOREM
    TT(G)(i)(e_Down)(1) < TT(r)(i)(e_In)(1) ∧
     TT(r)(i)(e_Out)(1) < TT(G)(i)(e_Up)(1)
 END railroad_safety
```

G.2 PVS Proof Commands for Safety Property of Railroad Crossing

```
(|model|)
(|transition_time|)
(|train|)
(|controller|)
(|gate|)
(|railroad|)
(|railroad_safety|
(|safety| "" (LEMMA "TR_AX_3")
  ((""  (LEMMA "TC_AX_1")
    ((""  (LEMMA "TC_AX_2")
      ((""  (LEMMA "TC_AX_3")
        ((""  (LEMMA "TC_AX_4")
          ((""  (LEMMA "TC_AX_5")
            ((""  (LEMMA "TC_AX_6")
              ((""  (LEMMA "SY_AX_3")
                ((""  (LEMMA "SY_AX_4")
                  ((""  (LEMMA "FirstTrain_TR_AX")
                    ((""  (LEMMA "LastTrain_TR_AX")
                      ((""  (LEMMA "FirstNear_TR_AX")
                        ((""  (LEMMA "LastExit_TR_AX")
                          ((""  (LEMMA "FirstNear_SY_AX")
                            ((""  (LEMMA "LastExit_SY_AX")
                              ((""  (SKOSIMP)
                                ((""  (INST -5 "i:1" "r:1")
                                  ((""  (INST -6 "i:1" "r:1")
                                    ((""  (INST -13 "i:1" "r:1")
                                      ((""  (INST -14 "i:1" "r:1")
                                        ((""  (INST -15 "i:1" "r:1")
                                          ((""  (GRIND) NIL NIL))
                                          NIL))
                                        NIL))
                                      NIL))
                                    NIL))
                                  NIL))
                                NIL))
                              NIL))
                            NIL))
                          NIL))
                        NIL))
                      NIL))
                    NIL))
                  NIL))
                NIL))
              NIL))
            NIL))
          NIL))
        NIL))
      NIL))
    NIL))
  NIL))
```