INFORMATION TO USERS

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

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

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

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

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

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

UMI®

AUTOMATIC CODE GENERATION FOR REAL-TIME REACTIVE SYSTEMS IN TROMLAB ENVIRONMENT

DA QING ZHANG

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 2001

© DA QING ZHANG, 2001

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-59345-2

Canada

Abstract

Automatic Code Generation for Real-Time Reactive Systems in TROMLAB Environment

DA QING ZHANG

Real-time reactive systems are among the most difficult systems to design and implement because of their complex functional and timing requirements. TROMLAB is a rigorous framework founded on TROM formalism for developing real-time reactive systems. The framework provides a number of tools that make formalism transparent to the application developer. The work presented in this thesis adds a new component to TROMLAB. The thesis gives a methodology for the implementation of a real-time reactive system designed and validated in TROMLAB. The methodology consists of defining a real-time execution model in C++ that fits the TROM formalism, automatically translating TROM specifications into Larch/C++, an interface specification language, and finally mechanically generating C++ code that conforms to the interface specification. The code generation methodology is illustrated for the *Train-Gate-Controller* problem, a bench-mark case study in real-time systems community.

Acknowledgments

I would like to deeply thank my supervisor, Dr. V.S.Alagar, who introduced me to the topic of the thesis, and guided me in developing the methodology to mechanical code generation. I deeply thank him for his kindness and patient reading of my thesis drafts. His understanding and support throughout the period of my work is greatly appreciated.

I also thank all members of TROMLAB research group for their technical support and friendship.

I thank my family for their understanding and support during my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Real-time Reactive System

Real-time reactive systems are largely event-driven, interact continuously with the environment through stimulus-response behavior, and the behaviors are regulated by strict timing constraints.

In general, the behavior of a real-time reactive system is infinite: a process in a real-time reactive system is usually non-terminating. In this respect, real-time reactive systems are different from common transformational systems, which can be regarded as functions from the input space at the start of a computation to an output space on termination. Real-time reactive systems are also different from interactive systems. The major difference between them is the availability of synchronization mechanism: a real-time reactive system is fully responsible for synchronization with its environment. Two important hypothesis for real-time reactive systems are as following:

- *stimulus synchronization*: the process always reacts to a stimulus from its environment;
- *response synchronization*: the time elapsed between a stimulus and its response is acceptable to the reactive dynamics of the environment so that the environment is still receptive to the response;

Examples of real-time reactive systems include air traffic control systems, nuclear reactors, avionic control systems, and communication network switches, etc.

Real-time reactive systems run in safety-critical contexts with concurrency, hard real-time requirements, large variations in timing regulations, and various possibility of system executions caused by the interleaving of concurrent activities. The functional behavior and real-time constraints on the actions must be analyzed to guarantee the satisfaction of safety requirements before the system brought into action. But the complexity of the interactions makes it difficult to comprehend or analyze the behavior of real-time reactive systems.

The largeness, criticality, concurrency, and the time-dependent behavior are some of the critical factors that contribute to the complexity of a reactive system. Large real-time reactive systems become more difficult to understand, maintain, and modify. A formal approach that can model the real-world entities of real-time reactive systems could help to reduce these difficulties. For this purpose, a formal specification methodology for real-time reactive system development, TROM formalism, was introduced by Acuthan[Ach95].

1.2 Contribution of the Thesis

Thesis topic arose in the context of TROMLAB and its work is based on the following ideas: (1) TROM formalism, which has object-oriented features, can be mapped into object-oriented language. (Here we select C++, an object-oriented language which is widely used, as the implementation language); (2) In the C++ implementation of real-time reactive system, each specific reactive object can be regarded as a thread in run time C++. The running of a real-time object reactive system, therefore, could be regarded as a multi-threaded execution.

The objective of my work is to give a methodology for the implementation, from design model in TROM formalism to C++. Based on the methodology, the code generation process of real-time reactive system is automated so that the resulting program is correct and efficient.

The main contributions of my work are the followings:

- Mapping TROM formalism into a C++ Implementation model;

- Implementation of TROM objects and subsystems;
- Implementation of real-time C++ model;
- Development of tools to generate Larch/C++ specification and C++ code;
- Generating the data type library in C++;
- Conformance evaluation of the generated C++ code to Larch/C++ specification.

Figure 1 shows an implementation model of the research components

The organization of this thesis is as follows. Chapter 2, introduces the TROM formalism, TROMLAB tools, and the significance of code generation. In Chapter 3, we discuss the object-oriented features of TROM formalism that need to be implemented in C++. We also provide a discussion on the advantages and defects of C++ for the implementation. Chapter 4 gives the mapping from TROM formalism to C++ class hierarchy. Chapter 5 gives the implementation of real-time C++ model. Chapter 6 presents how to implement the code generation—from TROM to Larch/C++ and C++ code. Chapter 7 shows the code generation steps for the *Train-Gate-Controller* case study. Chapter 8 gives the conclusions and the future work.

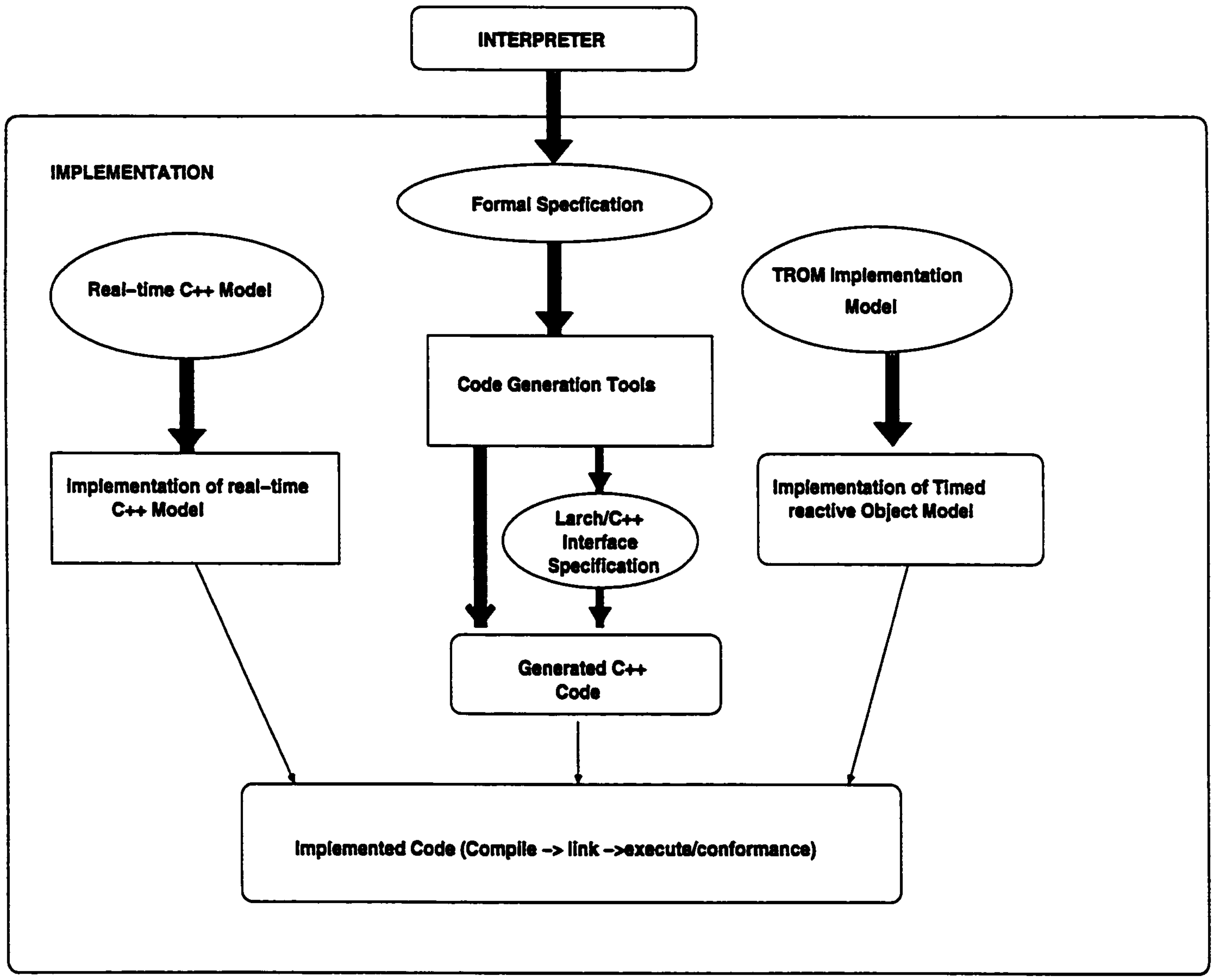

INTERPRETER
IMPLEMENTATION
Formal Specfication
Real-time C++ Model
TROM Implementation Model
Code Generation Tools
Implementation of real-time C++ Model
Larch/C++ Interface Specification
Implementation of Timed reactive Object Model
Generated C++ Code
Implemented Code (Compile -> link ->executa/conformance)

Figure 1: Illustration of the implementation

Chapter 2

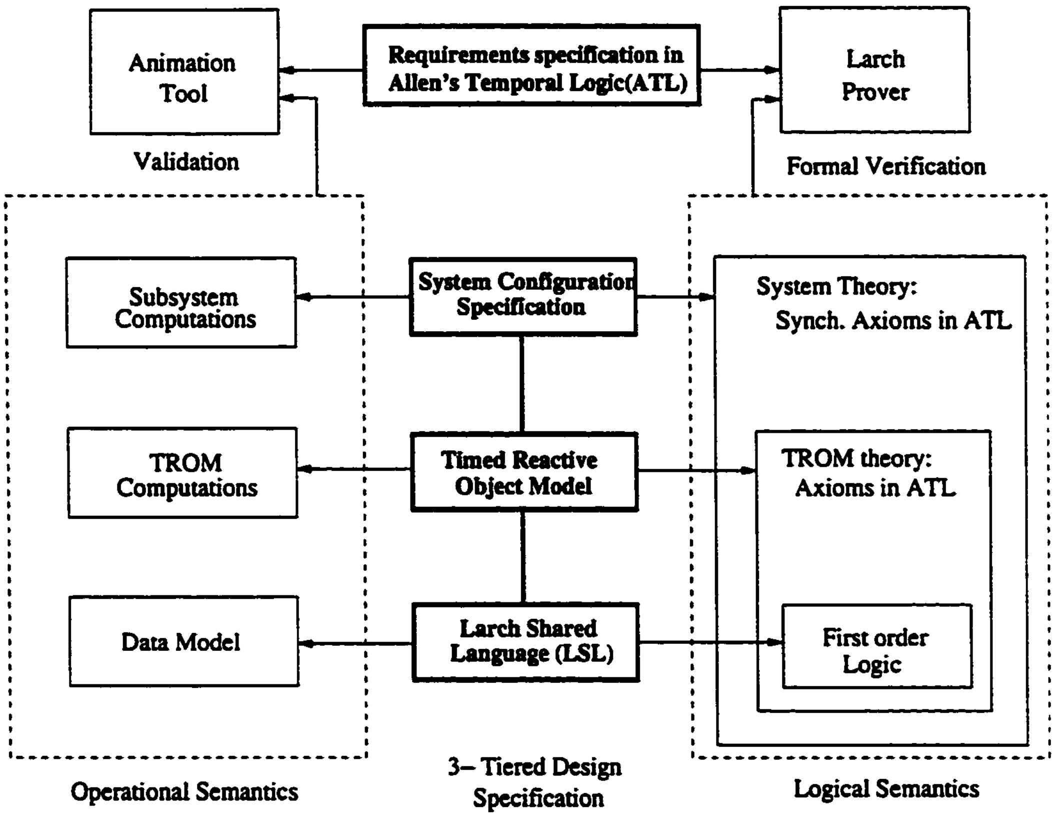
TROMLAB Environment

2.1 TROM Formalism

Achuthan [Ach95] introduced an abstract reactive system model, it has three tiers: (1) mathematical abstractions of data models used in specifying a reactive object, (2) reactive objects with time-constrained stimulus-response behavior, and (3) object collaborations. The three tiers independently specify abstract data types, generic reactive classes, and system configurations, respectively. Reactive class specifications include abstract data types specified as LSL (Larch Shared Language) [GH93] traits. A subsystem specifies instantiations of generic reactive classes, links to configure interaction channels among the objects, and compositions of subsystems. Figure 2 shows an overview of the TROM formalism.

In TROM formalism [Ach95], a reactive object is assumed to have a single thread of control. The communication mechanism is based on *synchronous message passing*. 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.

In TROM formalism [Ach95], a reactive object consists of port types, event, states, attributes, LSL traits, an attribute function, transition specifications , and time constraints. A states can be simple, or complex (with substates). The attributes function defines the association of attributes to states; A transition specification describes the



Animation Tool
Requirements specification in Allen's Temporal Logic(ATL)
Larch Prover
Validation
Formal Verification
Subsystem Computations
System Configuration Specification
System Theory: Synch. Axioms in ATL
TROM Computations
Timed Reactive Object Model
TROM theory: Axioms in ATL
Data Model
Larch Shared Language (LSL)
First order Logic
Operational Semantics
3– Tiered Design Specification
Logical Semantics

Figure 2: An overview of TROM Formalism

computational step associated with the occurrence of an event. 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 cause the constrained event to be enabled; the enabled reaction is disabled when the object enters one of the *disabling* states associated with the time constraint.

Formally, timed reactive object model (TROM) [Ach95] defining a reactive object, is an 8-tuple ($\mathcal{P}$,$\mathcal{E}$,Θ,$\mathcal{X}$,$\mathcal{L}$,Φ,Λ,Υ) such that:

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

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

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

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

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

  For each $\theta \in \Theta$, the silent-transition $\lambda_{s\theta} \in \Lambda$ is such that,
  $\lambda_{s\theta} : \langle\theta, \theta'\rangle; \textbf{tick}; true \Rightarrow \forall_x \in \Phi_a t(\theta) : x = x'$;
  The initial-transition λ_{init} is such that $\lambda_{init} : \langle\theta_0\rangle; Create(); \varphi_{init}$ where φ_{init} is an assertion on active-attributes of θ_0.

- Υ is a finite set of time-constraints. A timing constraint $v_i \in \Upsilon$ is a tuple $(\lambda_i, e_i', [l, u], \Theta_i)$ where,
  -$\lambda_i \neq \lambda_s$ is a transition specification.
  -$e_i' \in (\mathcal{E}_{out} \cup \mathcal{E}_{int})$ is the constrained event.
  -[l,u] defines the minimum and maximum response times.
  -$\Theta_i \subseteq \Theta$ is the set of states wherein the timing constraint v_i will be ignored.

Figure 3 illustrates the template for class specification of generic reactive class in TROM formalism. The keyword Class introduces a reactive class with its identifier and 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 generic reactive class.

```
Class < emidentifier > [< porttypes >]
    Events:
    States:
    Attributes:
    Traits:
    Attribute-Function:
    Transition-Specifications:
    Time-Constraints:
end
```

Figure 3: Template for Class Specification.

Figure 4 shows a TROM class description for specifying an *arbiter*. An arbiter allocates shared resources to processes requesting them. The arbiter modeled in Figure 4 enqueues the requests for a resource received from processes and allocates the resource to the next process waiting in the queue. The specification uses the functions *insert* and *tail* from the trait *Queue(@U, UQueue)*, imported into the *Arbiter* class, to add and delete requests made at a port of type *@U*. The attribute *hold* in class *Arbiter* denotes the most recent port at which the resource was granted. Input and output events are marked by the suffix symbols '?' and '!' respectively. Internal events are unmarked. The output event *Grant!* is time constrained and must occur within 2 units of time from the instant the input event *Req?* and *Ret?* have occurred.

In TROM formalism [Ach95], 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 this time constraint. The TROM object is in a *stable status* when the reaction

```
Class Arbiter [@U]
 Events: Req?U, Grant!U, Ret?U
 States: *idle, allot, wait
 Attributes: rqSet: USet; hold:@U
 Traits: Set[@U, USet]
 Attribute-function:
  allot ↦ {rqSet}; wait ↦ {rqSet, hold};
 Transition-Specifications:
  R1 : ⟨idle, allot⟩; Req?(true);
      true ⟹ rqSet′ = insert(pid, {});
  R2 : ⟨allot, wait⟩; Grant!(pid ∈ rqSet);
      true ⟹ rqSet′ = delete(pid, rqSet)) ∧ (hold′ = pid);
  R3 : ⟨allot, allot⟩, ⟨wait, wait⟩; Req?(not pid ∈ rqSet);
      true ⟹rqSet′ = insert(pid, rqSet);
  R4 : ⟨wait, allot⟩; Ret? (pid = hold);
      ¬ isEmpty(rqSet) ⟹ rqSet′ = rqSet;
  R5 : ⟨wait, idle⟩; Ret? (pid = hold);
      isEmpty(rqSet) ⟹ true;
 Time-Constraints:
  TC1 : (R1, Grant, [0,2], ∅)
  TC2 : (R4, Grant, [0,2], ∅)
end
```

Figure 4: Class definition of Arbiter

vector is null.

A *computational step* [Ach95] of a TROM object is an atomic step which takens the TROM object from one status to its succeeding status as defined by the transition specifications. Each computational step of a TROM object is associated with a transition in the TROM object; and each transition is associated with either an interaction signal, an internal signal, or a '*seilent* signal. A computation step occurs when the TROM object receives a *signal* and there exists a transition specification such that: the triggering event for the transition is the event causing the signal; the TROM object is in the source state or 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 reactive system (subsystem) is the set of statues of the TROM objects in the system (subsystem). And the computation of a TROM object is a sequence of computational steps.

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 that the event occurs is within the reaction windows 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 of the outstanding reaction, then the reactionis disabled. Whenever a reaction is time-constrained by the transition associated with the computational step, the reaction is enabled.

2.2 TROMLAB Tools

TROMLAB is a framework for applying TROM formalism and constructing real-time reactive systems in concordance with the process model.

Figure 5 illustrates the architecture of TROMLAB.

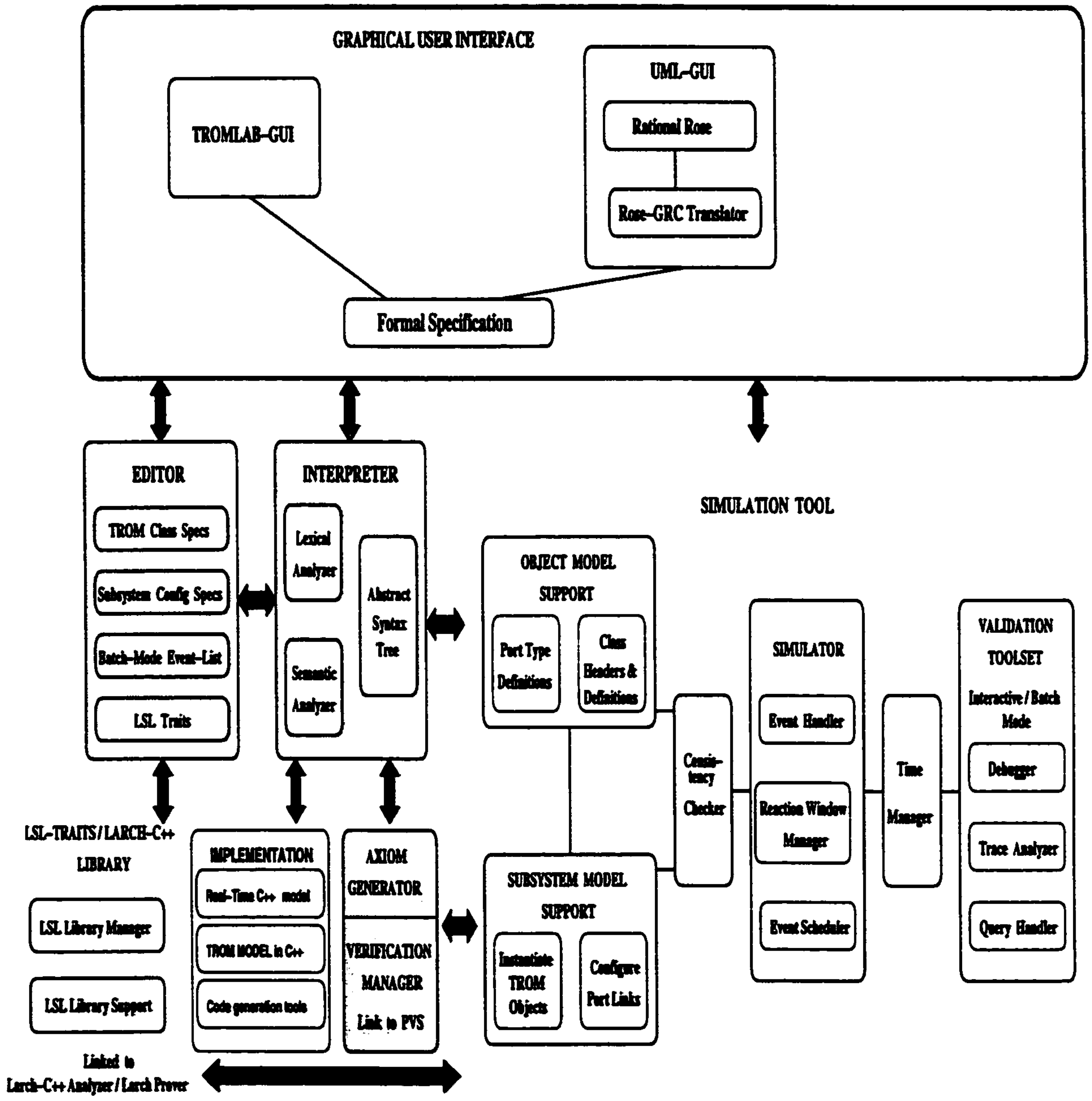


Figure 5: TROMLAB Architecture

The following components of TROMLAB are completed:

Interpreter [Tao96] parses, syntactically checks a specification, and constructs an internal representation;

Simulator [Mut96] animates a subsystem based on the internal representation, and enables a systematic validation of the specified system;

Browser for Reuse [Nag99] which is an interface to the library, to help users navigate, query and access various system components for reuse during system development;

Graphical User Interface [Sri99] is a visual modeling and interaction facility for a developer using the TROMLAB environment;

UML-RT Support [Oana99] is a translator to generate TROM specification from real-time UML;

Verification Assistant [Pop99] is a tool to generate PVS theory from TROM specification for proving timing properties;

Reasoning System [Hai99] provides a means of debugging the system during animation by facilitating interactive queries of hypothetical nature on system behavior.

Alagar and Muthiayen [AM98] proposed minimum extension to Unified Modeling Language(UML) notations to model real-time reactive systems as conceived in the TROM formalism, and gave the research work [Mut98] in integrating object-oriented methodology as practiced through the UML notations in industries and formal verification approaches based on the formal specification language Prototype Verification System(PVS) [ORS92].

2.3 Significance of Automatic Code Generation

Automatic code generation from formal reactive system design is a new area of research. Given that real-time reactive systems are time-critical, efficiency of implementation is more important than in other systems. Responding to situations where all the timing constraints cannot be met and meeting the timing requirements that may change dynamically are two of the most difficult issues to satisfy in an implementation. In practice it is impossible to design and implement systems which will guarantee that the appropriate output will be generated at the appropriate time under all possible conditions. The accuracy of modeling environment constraints and real-world entities that interact with the system is crucial in an implementation. In particular, when the system is large, the implementation will become more difficult. It is necessary to give an efficient methodology for system design as well as a generic approach from design model to the implementation of the system. The work accomplished so far in TROMLAB framework provides a setting for implementation on a design that has been validated and verified. That is, at the design level timing analysis and functional behavior are conducted prior to committing to an implementation. Moreover, the iterative cycle in the TROMLAB process model allows faulty implementations to go through an inner cycle of redesign, validation, and verification. This improves the level of reuse of specifications, design, and implementation modules.

In addition, the approach of the implementation and code generation gives a facility to verify whether the functional and timing constraints in the implemented program are consistent with the implicit functionality specified in the transition specifications and the timing constraints specified for the time-constrained events. A good implementation must seamlessly mesh with the design. This quality depends upon the expressive power of the programming language that used to implement the design notations. In the following Chapter, we will discuss the advantages and defects in C++ for this implementation, together with the conformance of the generated code to design notations.

Chapter 3

Object-Oriented Concepts in TROM

3.1 Object-Oriented Concepts in Real-time Reactive System

One of the important features of the TROM methodology is the conciseness achieved in specification with object-oriented concepts such as encapsulation, inheritance and subtyping. Moreover, the three-tiered design framework provides a clear separation of concerns for system refinement and promotes reuse.

Consider the application of data abstraction in real-time reactive system. Larch traits can be developed incrementally by including one trait in another. The components in the second and third tiers of a design framework can use abstract types from the lowest tier in a black-box fashion. That is, the interface between tiers completely insulates the implementation from its details. This is advantageous to real-time reactive system because detailed information concerning data members of objects, such as resources, need not be known until later stages of design.

A reactive object is a black-box which can respond to a fixed set of messages from outside by carrying out some of its operations. An object combines state and behavior in a single encapsulated entity. The property of encapsulation essentially means that an object's state may change only as a result of internal actions or as a result of interaction with the environment through explicitly defined external interface operations. The object-oriented concepts relevant to this methodology are defined and interpreted in the context of specifications.

Objects communicate by messages sent through the ports associated with them. Communication mechanism between objects is based on synchrony hypothesis. That is, an object that sends the message $e?$ and the object that receives $e!$ change their states *simultaneously*. The synchrony hypothesis assumes that computation times are null and transition is instantaneous.

A subsystem is an aggregation of objects. Objects in a subsystem interact through messages sent/received along the portlinks that connect the compatible ports in the interaction. While two objects are interacting, other objects in the system may perform internal actions. Thus, system may execute concurrently. A subsystem may include other subsystems. That is, subsystems provide a coarser level of encapsulation and help to structure large systems.

Inheritance is used as a kind of design technique by which one class definition is derived incrementally from one or more class definitions. In TROM formalism, inheritance is used for the refinement of system design, which add more details, such as more *states*, more *transition*, and strengthening *time constraints*. The refined design should preserve essential properties of the original design and aid the implementation process, but the design from an unconstrained inheritance principle cannot guarantee the preservation. For this reason, three forms of constrained inheritances based on *subtyping* were introduced in TROM formalism. They are *behavioral inheritance*, *extensional inheritance* and *polymorphic inheritance*.

TROM formalism supports inheritance by which a type (new class) can be derived from one or more existing direct base class (single inheritance or multiple inheritances). Inheritance in object oriented paradigm defines new classes using the classes that already exist in the system, and only the properties of the new classes that differ from the existing classes should be defined explicitly, while other properties will be inherited from the existing classes. The utilization of inheritance exists between the class hierarchies. With this feature, the abstract classes are built as the generic model,

Inheritance	*Behavior-Preservation*	*Substitutability*
Behavioral	*yes Context – independent*	*yes Context – independent*
Extensional	*yes Context – independent*	*no*
Polymorphic	*yes Context – dependent*	*yes Context – independent*

Table 1: Features of different kind of inheritance supported in TROM

and the classes of a special case inherits the properties from the abstract classes.

Inheritance that is behavior preserving is referred to as subtype-inheritance. The subtype-inheritance can be of different kinds whereby substitution polymorphism or behavioral extensions are achieved. This provides the basis for incremental and iterative system development in the framework.

Polymorphism is the notion of behavioral compatibility used as a design technique by which an object can be substituted by another compatible object. Class hierarchies with polymorphism through inheritance are of special interest for both system design (system enhancement) and implementation (instantiation).

3.2 C++ Implementation of Real-time Reactive System

C++ lacks the ability to react to external stimuli and often cannot describe concurrency. For this reason, when it is required to implement a real-time reactive system, an extension to C++ should be developed. The goals are to be able to have different concurrent modules, to describe their interconnections and provide an event-driven communication scheme, with methods to suspend and resume a task, or abort a certain task when a given condition occurs.

Different from Java (another popular object oriented language), portability in C++ is an issue; it is because portability is taken into account during the design of Java language, but it is not so for C++. Moreover, as an object oriented language evolved from C, there are some problems of historical legacy that plague C++.

The syntax in C++, specially in the expression of abstract data type and its behavior as well as some mathematical theories, is different from the TROM formalism which expresses its data model by Larch Shared Language based on first-order logic. Besides, C++ lacks the strong capability to express time properties. The difference in expression logic between C++ and the specification language may augment the risk of inconsistency in the C++ implementation and its formal specification.

The standard C++ does not provide multi-threaded mechanism for object oriented programming. In the implementation of real-time reactive system, the mechanism should be developed on specified platform and based on some features of the specified C++ version. In Chapter 5, the implementation of the mechanism is given.

Even if some extension in C++, that makes a program react to stimuli, can be achieved by using the developed thread synchronization, as the number of signal increases, it sometimes becomes less tractable, in particular, when more than one inputs are expected at the same time. For this cause, there are also several problems with the thread mechanism in general, and with its implementation in C++ also.

Some of the problems that may arise due to a "weak" thread mechanism can be listed as follows:

1. Provides designers with a low-level control of parallelism and a great deal of freedom for developing parallel applications, which is usually not required, and often leads to bad designs with difficult-to-find bugs;

2. Lacks control structures for communication between threads; and

3. Lacks a standard scheduling algorithm for different environment

We will discuss solutions to these problems in the following Chapters.

Chapter 4

TROM Implementation Model

4.1 Abstract TROM Class

In TROM formalism, a reactive object is regarded as a finite state machine, extended with attributes. TROMs interact synchronously by signaling output events and by responding to input events signaled in their ports. The reactive behavior of a TROM is controlled by time-constrained causal relationship between the events and the computations they trigger. Among its outstanding features, a TROM allows concise specification of encapsulated time-constraints, and allows object modeling at various levels of abstraction.

A reactive system is modeled as a collection of interacting reactive objects. The specification of a reactive system is made from the specifications of individual reactive objects and their collaborations. An object is the smallest unit of encapsulation with a single thread of control. Each object has its unique identifier.

In the implementation, Class TROM, as the basic type of reactive system, is defined as the concrete type of C++, with the data member of *ports*, *states*, *events*,*attributes*, *transitions*, and *constraints*. It is defined as a derived class of Class Thread.

A *generic reactive class* (GRC), as a derived class of Class TROM, models the structure and behavior of a family of real-world reactive objects.

TROM(abstract TROM class)
set⟨Port⟩ ports;
set⟨Event⟩ events;
set⟨State⟩ states;
set⟨Attribute⟩ attributes;
set⟨Transition⟩ transitions;
set⟨Constraint⟩ constraints;
string name;
deque ⟨ event ⟩ eventq;
int localclock;
TROM();
~TROM() ;
void SetName(string nm);
string GetName();
string ConnectObj(string pn);
bool TimeConstraint(string ename);
string GetType(string ename);
string GetPortName(string ename);
bool EventNameInEvents(string ename);
bool PortNameInPorts(string pname);
void* Run(void* buf);

Table 2: The Class TROM

Event(abstract event class)
string eventname;
string eventtype;
string porttype;
Event();
~Event();
Event(string name,string type,string ptype);
void SetEventName(string ens);
string GetEventName();
void SetEventType(string t);
string GetEventType();
void SetPortType(string pas);
string GetPortType();

Table 3: The Class Event

4.2 Events

Because reactive objects are event-driven, it is possible to describe them in terms of their dynamic behaviors. The specification of a reactive object is succinctly captured by a set of behaviors, where each behavior is a trace of massage passing that occurs at the interface between the object and its environment.

As a group of threads, objects need to synchronize their activities to effectively interact. The approach for synchronization includes: implicit communications through the modification of shared data and/or explicit communications by informing each other of events that have occurred.

In the implementation, the communications are realized by the way of modifying shared data space—*event_buffer*.

Class Event defines an event with the properties of the name of the event: *eventname*, the type ("input", "output", "internal" or a silent event) of the event: *eventtype*, and the port type associated with this event: *porttype*.

4.3 States and Attributes

A TROM, at any time, is in an abstract state. Each state in a TROM abstractly represents a collection of properties that is held when the TROM is in this state. There is a set of attributes that are associated with each state. States signify the *qualitative* aspects of the behavior of an object, while the attributes signify the *quantitative* aspects. A change in value of an attribute does not mean a change of state, and attributes do not need to have different values in different states, as well.

The state in a TROM can have *substates*, and therefore, can be hierarchical. Hierarchical state helps to control the complexity of specification by structuring the behavior into sets of simpler units which could be comprehended individually. Any property associated to a state will be inherited in all of its substates. Furthermore, there are great advantages in selecting the right level for a detailed specification.

A set of states, together with a set of attributes, are encapsulated in the generic reactive class, and they can only be changed by computations local to it . A computation usually involves a state transition and an assignment of values to attributes. Computations are triggered by events, and may be required to meet the requirement of time constraints. At any instant, the state and its attribute values determine whether or not a TROM undergoes an interaction, at a particular port and with a particular object in its environment.

Class State defines a state with the properties of the state name: *statename*, an adjustment of either an initial state or not: *initialstate*, and the *substates* of the state.

Class Attribute defines a kind of attribute with the attribute name: *attributename* and the attribute type: *attribute type*.

The abstract data types of the value in attribute are introduced by LSL traits.

If the attribute type is a port type, *porttype* will be used to set its value. when the attribute type is a trait, *trait* will be used to set its value.

State (abstract state class)

string statename;
bool initialstate;
set ⟨ state ⟩ substate;

State();
~State();
State(string name,bool ini, set< *state* > subs);
bool Atomic();
void SetStateName(string n);
string GetStateName();
void SetInitial(bool i);
bool GetIinitial();
void SetSubs(set⟨state⟩ s);
set ⟨ state ⟩ GetSubs();

Table 4: The Class State

Attribute(abstract attribute class)

string attributename;
bool attributetype;
string trait;
string porttype;

Attribute();
~Attribute();
Attribute(string na,bool t,string tra,string p);
void SetAttributeName(string n);
string GetAttributeName();
void SetAttributeType(bool t);
bool GetAttributeType();
void SetTrait(string tra);
string GetTrait();
void SetPortType(string p);
string GetPortType();

Table 5: The Class Attribute

Port(abstract port class)

string portname;
string portype;
string objectname;
string conportname;
string conportype;
string conobjname;

Port(string tp);
~Port();
Port(string na,string tp,string obn,string cna,string ctp,string cobn);
void SetPortName(string n);
string GetPortName();
void SetPorType(string t);
string GetPorType();
void SetObjectName(string ct);
string GetObjectName();
void SetConPortName(string cna);
string GetConPortName();
void SetConPortType(string ctp);
string GetConPortType();
void SetConObjName(string cobn);
string GetConObjName();

Table 6: The Class Port

4.4 Ports

A port is an abstraction of an access point for a bidirectional communication channel between a TROM and its environment.

Each port has a unique port-type.

A port-type dictates the set of message and the possible message sequences that are allowed at a port of that particular type.

A TROM can have multiple port-types associated with it, and a TROM can also have multiple ports of the same type associated with it.

A null-port, denoted by o, is associated with every TROM.

In the C++ implementation, port-type is a property concerned with each port.

Two port types P and Q are compatible if

- $e? \in \mathcal{E}_i n^P \Leftrightarrow e! \in \mathcal{E}_o ut^Q$
- $e! \in \mathcal{E}_o ut^P \Leftrightarrow e? \in \mathcal{E}_i n^Q$

Class Port defines an port with port name: *portname*, port type: *port type*, *name of the object* which is with this port: objectname, the connecting port name: *conportname*, the connecting port type *conportype* and the connecting object name *conobjname*.

4.5 Transitions

Transitions are labeled by events, which are the message components used to capture the interaction of an object with its environment. The attributes of a state not only enhance the expressive power of the model in a concise way, but also help to model the data computations associated with transitions. The abstract behaviors of attributes are specified in a definitional style using LSL.

Assertions associated with each transition are:

1. *enabling-condition*, specifying the conditions to be satisfied for the transition to be initiated,
2. *post-condition*, specifying the data computation associated with the transition,
3. *port-condition*, specifying the port at which an interaction can happen. The associated LSL traits provide the vocabulary for these assertions.

Class Transition defines a transition with the source state *sstate*, destination state *dstate*, the trigger event of the transition: *triggername*, and the timer *timer* to record the time, together with the *enabling-condition* of the transition, the *port-condition* of

Transition (abstract transition class)

string transitionname;
string triggername;
string sstate, dstate;
int timer;

Transition();
~Transition();
Transition(string n, string en, string ss, string ds);
void SetTransitionName(string n);
string GetTransitionName();
void SetTriggerName(string en);
string GetTriggerName();
void SetSourState(string s);
string GetSourState();
void SetDestState(string s);
string GetDestState();
void SetTimer(int i);
int GetTimer();
virtual bool port_condition(string pid) = 0;
virtual bool pre_condition(string pid)= 0;
virtual void effect() = 0;
virtual bool post_condition(string pid) = 0;

Table 7: The Class Transition

the trigger event, and the *post-condition*. The predicates are defined as pure virtual functions, so the class is of polymorphic type.

A *generic transition class* (GTC) is the derived class of Class Transition, in which the virtual functions from Class Transition must be defined the details.

4.6 Time Constraints

The timing constraints define the reactions of a TROM. Every reaction has an associated trigger which corresponds to a transition – that is, the occurrence of an event when the TROM is in a certain state. This means that the occurrence of an event may or may not be a trigger depending on the state of the TROM. Any reaction of a TROM can involve only one output event or one internal event. It is because the input events are not under the control of the concerned TROM, instead they are

Constraint (abstract time_Constraint Class)

string constraintname;
string tranname;
string conseventname;
int mintime,maxtime;
set ⟨ string ⟩ distates;

Constraint();
~Constraint();
Constraint(string na,string tr,string en,int mi,int ma, set< *string* > s);
void SetConstraintName(string n);
string GetConstraintName();
void SetTransitionName(string t);
string GetTransitionName();
void SetConsEventName(string en);
string GetConsEventName();
void SetMinTime(int i);
int GetMinTime();
void SetMaxTime(int a);
int GetMaxTime();
void SetDisableStates(set ⟨ string ⟩ s);
set ⟨ string ⟩ GetDisableStates();

Table 8: The Class Constraint

determined by the TROM's environment.

The Class Constraint defines a time-constraint with the constrained transition name *transname*, the constrained event name *conseventname*, the bounds of response times: *mintime* and *maxtime*, and the set of states where in this timing constraint will be ignored: *distates*.

Chapter 5

Real-Time C++ Model

5.1 Multi-threaded Mechanism

Multi-threaded programming as an alternative to multi-process programming, is less demanding of system resources. In my implementation, the collection of reactive objects is developed as a group of threads within a single process. The individual threads can be regarded as running concurrently and need not implement task switching explicitly, instead it is handled by the operating system and the multi-threaded mechanism developed.

In the implementation, one of the threads in the process is the *primary thread*, it corresponds to the lone thread in a single-threaded program. This primary thread is activated when the operating system invokes the *main()* function, and then it creates additional *secondary threads* which run as the reactive objects. Each of the secondary threads executes its own *thread function*, just as the *main()* of the primary thread. The thread functions is passed initialization parameters. All the threads within the process share the memory allocated to the process. Therefore, the reactive objects can communicate directly through ordinary program variables.

The principal approach for communications between reactive objects is by modification of the shared process environment. However, the scheduler can interrupt the execution of a thread at unpredictable instants and this opens the possibility that one thread may not leave the process environment in a consistent state ready for the next thread to be activated. The solution to this problem is to apply the technique of *thread synchronization* to regulate the activities of the interacting reactive objects. Three basic techniques are generally used in the threads synchronization: *mutexes*, *condition variables*, and *joining*. More complex synchronization can be built using the primitive approaches.

A mutex is used to provide mutually exclusive access to a shared resource. The resource will be assigned a mutex to protect it. Before a reactive object is admitted to manipulate the resource, it must acquire the mutex. No other reactive object can access the resource when the mutex is held. After the reactive object using the resource finishes, the associated mutex will be released, and other reactive objects can compete to acquire it.

The C++ language supports object-oriented programming. A reactive object in C++ comprises a data structure describing the object's static property together with a collection of functions that define the capabilities of the object. The *current state* of the reactive object determines the exact process to be performed by it whenever the functions are invoked. In the implementation, each reactive object, as an object in C++, belongs to a particular C++ class and all of these objects in this class share the same set of functions. The individual objects within the class are distinguished by their own independent data.

An event is either *send or receive* between reactive objects through a buffer. The buffer, as shared environment, plays a role of handling the event sent. When an event on a specific port, as a trigger of a transition of an object, is available in the buffer, the transition of this object will be triggered if other requirements (i.e., the time constraints) are met. When a thread is invoked, a mutex as a lock is used to avoid other threads from accessing shared resource.

From the point of view for scheduling, the computation of a real-time reactive system here can be thought as an instance of the distributed computation, which admits solution that contains a phase where some global or local property needs to be detected. The ability to construct a "global state" of the system and evaluate a predicate over such a "global state" constitutes the core of the solution.

A solution for the reactive system scheduling is described as follows: each reactive object is allotted a time slice, after which it is preempted to allow others to run.

Based on this scheduling algorithm, buffer is used as a shared storage to record the event sent or received through ports.

The advantage of developing and using the thread mechanism instead of a normal serial program is that several operations may be carried out in "parallel", and thus events can be handled "immediately" as they arrive.

The advantage of using a thread group over using a process group is that context switching between threads is much faster then context switching between processes. Also, communications between two threads are usually faster and easier to implement than communications between two processes.

On the other hand, just as we discussed above, threads in a group all use the same memory space. Without a synchronization mechanism, whenever a thread corrupts the contents of its memory, other threads might suffer as well. But the situation is different for processes; the operating system normally protects a process from others, and thus if one process corrupts its own memory space, other processes would not suffer. Another difference between them is that, processes can run on different machines, while all the threads normally run on the same machine.

In my implementation, the abstract data types of TROM as well as the real-time C++ model, was developed, which make it possible to run a case of real-time reactive system when it is generated within C++ code. Besides that, the code generation tool, which generates the C++ code of a case from its formal specification, was implemented.

The principal application of thread synchronization here is to ensure that each modification of the process state is complete before control can be passed to another thread. The development and use of real-time C++ model is for meeting the two *synchronization requirements* of real-time reactive system.

C++ does not supply synchronization mechanism among different threads when accessing shared resources, neither it provides the multi-threading mechanism for object oriented programming. For this reason, it is necessary to implement these mechanisms for the real-time C++ model.

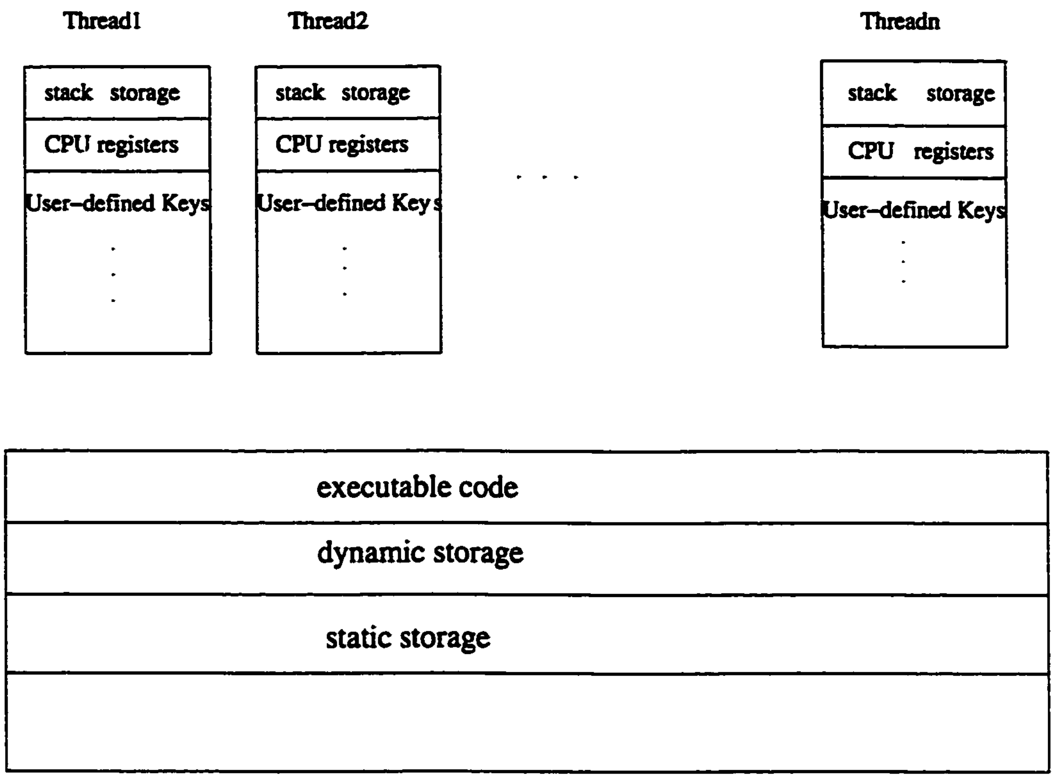


Figure 6: Shared resources and thread-specific resources

5.2 Shared Resources of Multiple Threads

A group of threads, as a collection of reactive objects, is created in a process environment. Part of this environment is shared by all threads in this process, while the rest is specific to individual threads. The situation is illustrated in Figure 6.

The shared environment of the reactive system comprises executable code, static storage and dynamic storage; the thread-specific context comprises stack storage, CPU registers and user-defined keys.

Because all reactive objects share the program's executable code, it is possible that more than one thread execute the same piece of code at the same time.

The remaining elements of the environment principally contain data. To manage the manipulation of these data structures by a collection of interacting threads is the key problem for these reactive objects running successfully.

Data in static storage is a globe variable in program, it exists all the time when the program running. Static storage is shared. When several reactive objects access the same data in static storage, they must synchronize their activities to avoid data corruption. When a reactive object is running, it will hold the storage and visit it randomly. No other reactive objects can access the buffer at this time.

Data held in a thread's stack storage is specific to an individual reactive object, the stack storage forms part of the thread context and is switched when a new reactive object is allocated a time-slice. When a reactive object makes a function call, space is allocated from the thread's stack storage to hold the *stack frame* for the function. The stack frame contains function parameters, local variables declared in the function and the address of the code that invoked the function, so that the reactive object knows where to resume execution when it returns from the function.

Because the stack storage is a part of each thread context, different reactive objects can simultaneously execute the same function, without fear of a thread modifying the function parameters and local variables of other threads.

Data in the CPU registers is also a part of the thread-specific context, and it is updated during a context switch. The scheduling algorithm may interrupt an active thread (the running of a reactive object) at any point in its execution so the next program instruction to be executed by this thread should be remembered, the relevant address in the program code is held in a CPU register.

Using the C++ **new** operator, data held in dynamic memory is allocated, which returns a pointer to the allocated memory. The memory can be deallocated by applying the **delete** operator to this pointer.

Dynamic data belongs to either the shared environment or the thread-specific context. This depends on the pointer. If the pointer is held in static storage then the dynamic memory is a shared resource; if the pointer is held in a thread's stack then the dynamic is thread-specific.

In the example of Figure 7, the *event_buffer* pointer references a double-ended queue which can be shared by other reactive objects. But the *local_buffer* pointer is in the stack frame of the current thread and references a buffer which is specific to this reactive object. The *local_buffer* pointer will be lost when the function returns, so the double-ended queue which it references must be deleted within the function.

```
set⟨event⟩* event_buffer;

void Function(void) {
event_buffer = new set ⟨event⟩;
set⟨event⟩ local_buffer = new set⟨event⟩;
.
.
.
static_cast ⟨ set⟨event⟩* ⟩ (event_buffer) → push_front(trigger);
.
.
.
delete local_buffer;
};
```

Figure 7: A Example of Dynamic Data

This technique is used when allocating large temporary structures. The advantage of using a local variable is that only the pointer appears in the function stack frame and the object is allocated in dynamic storage. Stack storage is smaller than the space available for dynamic storage.

These utilizations of dynamic storage allow for either shared data or thread-specific data. However, to provide long-lived thread-specific dynamic data, the thread-context may be extended by creating user-defined keys, which are essentially static pointers to thread-specific dynamic memory.

5.3 The Class Thread

As in Figure 8, Class Thread is defined to provide generic support for reactive objects.

Figure 8 defines the interface used by external code to work with threads. The internal implementation of the Class Thread depends on the operating system.

The *Start()* function of Class Thread is called to start the thread associated with a Thread object. Each Thread object corresponds to one reactive object. The *Start()* function invokes the *Run()* function which acts as its *thread function*, defines all

```
class Thread {
    public:
        Thread(void);
        virtual ~Thread(void);
        int Start(void*=NULL);
        void Detach(void);
        void* Wait(void);
        void Stop(void);
        unsigned int GetThreadID(void);
        static unsigned int GetCurrentThreadID(void);
        static void Sleep(int);
        .
        .
        };
```

Figure 8: The Class Thread

processing performed by the reactive object. The *Run()* function receives the **void*** parameter which is originally passed to *Start()* and this may be used to individualize the actions of different reactive objects.

In general, when *Run()* function eventually completes its processing, the function returns a **void*** value and the thread enters the *terminated* state where it remains until destroyed. Another thread may wait for the thread's return value by calling *Wait()* on the associated Thread object. If the return value is unwanted, the *Detach()* function should be called to allow the operating system to destroy the thread as soon as it terminates. The *Detach()* function can be called before or after *Start()*. Once a thread has been detached, the *Wait()* and *Stop()* functions will not work. The *Stop()* function is provided to abruptly terminate a thread but, if possible, its use should be avoided in favour of more controlled methods. Because the behavior of a reactive object generally is infinite, in the implementation of a real time reactive system, the thread as a running of reactive object will never be terminated.

Each reactive object as a thread, may have a unique ID which can be retrieved by calling the *GetThreadID()* function of the associated Thread object. The function *GetCurrentThreadID()* is static. It can be called without reference to a particular Thread object. This function returns the ID of the reactive object that calls the function. The *Sleep()* function is provided to allow a reactive object to pause its execution for a specified period of time, this function will move the threads to the *sleeping* state.

5.4 Implementation on UNIX

In this section, we give the implementation of the Class Thread on a UNIX system which supports the POSIX (*Portable Operating System Interface for UNIX*) standard, e.g. Solaris. It supports multi-threading functions such as the *pthread_create()* function.

The Class Thread under UNIX is shown as the header file in Figure 9. Under UNIX, the Thread class acquires some additions. Similarly, to implement the Class Thread on Windows, some additions are required as well.

The *ThreadFunction()* function is called by *Start()* and then it calls *Run()*. It is required just for the reason that: the thread creation function provided by Unix must be supplied with a global thread function, rather than that belongs to a C++ class. The *ThreadFunction()* function is a **friend** of the Class Thread so that it can invoke the **protected** *Run()* function; otherwise the *Run()* function is only available for the Class Thread and its derived classes. The *ThreadFunction()* function is declared as **static** to avoid being called by external code. The reactive object associated with the Thread object is identified by both a *ThreadHandle* as well as a *ThreadID*. The former is an opaque structure used only within the Class Thread while the latter is simply an **unsigned** integer retrievable with the function *GetThreadID()*. The *pthread.h* header file gives all the necessary declarations for working with POSIX threads.

The constructor function *Thread()* initializes variables when the Thread object is created. The destructor function ~*Thread()* is invoked when an object is no longer required.

Using the *pthread_create()* function we create a new thread which corresponds to a reactive object. The newly created thread starts to execute the function *ThreadFunction()* and receives a pointer to the reactive object as a parameter. This pointer is used to call the object's *Run()* function with the parameter which is originally passed

```
#include "unix.h"
#include ⟨ pthread.h ⟩

static void* ThreadFunction(void*);
class Thread {
friend void* ThreadFunction(void*);
public:
      Thread(void);
      virtual ~Thread(void);
      int Start(void*=NULL);
      void Detach(void);
      void* Wait(void);
      void Stop(void);
      unsigned int GetThreadID(void);
      static unsigned int GetcurrentThreadID(void);
      static void Sleep(void);
protected:
      virtual void* Run(void*);
private:
      pthread_t ThreadHandle;
      unsigned int ThreadID;
      };
```

Figure 9: Implementeation Thread

```
void* Thread::Run(void* param) {

return NULL;
};
```

Figure 10: The Function *Run()*

to the *Start()* function.

The virtual function *Run()* will be redefined in derived class, to give the process of reactive objects.

The *Wait*() function automatically detaches the thread. It uses the *pthread_join()* function which provided by POSIX standard.

The *Stop*() function abruptly terminates the thread if it is still running and has not previously been detached. Using the *pthread_cancel()* and *pthread_detach()* functions, it can be implemented under POSIX.

The basic mechanism we need, as shown above, can be developed under POSIX standard. But it is hard to limit whole the implementation within POSIX. At first, the POSIX standard does not give direct mechanism for the implementation of the *ThreadID* field as well as the functions *GetThreadID()* and the function *GetCurrent-ThreadID()*.

The *Sleep()* function can be implemented by using the *select()* function, as illustrated in Figure 11. The *select()* function is not in the POSIX standard, but it is commonly available on UNIX system.

The mutex functions *Demand()* and *GiveUp()*, are implemented by system function *pthread_mutex_lock()* and *pthread_mutex_unlock()*. The function *Demand()* creats a mutex to hold the resource. The function *GiveUp()* kills the mutex to unlock the resource. The *pthread_mutex_lock()* and *pthread_mutex_unlock()* functions are available under POSIX standard.

```
void Thread::Sleep(int delay) {

timeval timeout = {(delay/1000),((delay*1000)

select(0, (fd_set*) NULL,(fd_set*) NULL, (fd_set*) NULL,&timeout);
};
```

Figure 11: The Function *Sleep()*

Chapter 6

Code Generation Methodology

6.1 From TROM to Larch/C++

In traditional object-oriented approach, a system is developed in two levels: the first level is to define classes, and then the second level to instantiate the classes to compose subsystems. The TROM methodology adds a level for defining data models with Larch Shared Language. Thus, the TROM formalism has three levels: data models, object models and subsystem models. Here, object models use abstractions of data models, and subsystem models use instances of object models. It is to say that the data models of TROM formalism are based on Larch Shared Language.

Larch provides a two-tiered approach to the specification of program interfaces: the basic constructions are specified in shared tier and programming details are specified in the interface tier. In the shared tier, Larch Shared Language (LSL) is used to specify state independent, and mathematical abstraction that will be referred to in the interface tier. The unit of encapsulation in LSL, *trait*, introduces some operators and specifies some of their properties.

Larch/C++ is an interface specification language of Larch. In my implementation, the TROM transition specifications are at first automatically translated into Larch/C++ interface specifications. The C++ code of the transitions are generated automatically from their Larch/C++ interface specifications. Larch/C++, based on logic (LSL), has the features of C++ as well. So the conformance from the program generated in C++ to the interface specification in Larch/C++ is more easy to be given. This is why Larch/C++ is introduced in the implementation.

In Larch/C++ interface specification, the interface is given on the first line; And the body, its behavior, uses a group of predicates to describe the effect of the function invocation. The **requires** clause defines constraints on the state and parameters at the instance of the function invoked. The **modifies** and **ensures** clauses describe the behavior of the function while it is invoked properly. If a function is called when the program state satisfies the predicate in the **requires** clause, it will terminate in a state that satisfies the predicate in the **ensures** clause, and only those visible objects listed in the **modifies** clause can be changed.

In TROM model, the *Transition-specifications* is formally defined as following:

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

  -$\langle\theta, \theta'\rangle$,where $\theta, \theta' \in \Theta$ are the source and destination states of the transition, respectively.
  -$e(\varphi_{port})$ where event $e \in \mathcal{E}$ labels the transition; φ_{port} is an assertion on the attributes in $\mathcal{X}$ and a reserved variable **pid**. **pid** signifies the identifier of the port at which an interaction associated with the transition can occur. If $e \in \mathcal{E} \cup \{\textbf{tick}\}$, then the assertion φ_{port} is absent and e is assumed to occur at the null-port o.
  -$\varphi_{en} \Rightarrow \varphi_{post}$, where φ_{en} is the enabling condition and φ_{post} is the post-condition of the transition. φ_{en} is an assertion on the attributes in $\mathcal{X}$ specifying the condition under which the transition is enabled. φ_{post} is an assertion on the attributes in $\mathcal{X}$, primed attributes in $\Phi_{at}(\theta')$ and the variable **pid** specifying the data computation associated with the transition.
  For each $\theta \in \Theta$, the silent-transition $\lambda_{s\theta} \in \Lambda$ is such that,
  $\lambda_{s\theta} : \langle\theta, \theta'\rangle; \textbf{tick}; true \Rightarrow \forall_x \in \Phi_a t(\theta) : x = x'$;
  The initial-transition λ_{init} is such that $\lambda_{init} : \langle\theta_0\rangle; Create(); \varphi_{init}$ where φ_{init} is an assertion on active-attributes of θ_0.

In my code generation methodology, the predicates in the **requires** clause of a transition function specification refer to the assertions φ_{port}, φ_{en}, and *currentstate* $=$ *sourcestate* θ, these are the pre-condition that must be satisfied to invoke the function. The predicate in the **ensures** clause of the transition function specification refers to the assertion φ_{post}, the post-condition that the function establishes upon termination.

The semantics of transition is as follows: the pre-condition of this transition must logically imply the post-condition of the transition. Figure 12 illustrates the template for Larch/C++ transition interface specification.

```
void ⟨ Transition's Name ⟩ (Parameter Table)
behavior
    {
    requires φ_port; φ_en; currentstate = sourcestate θ;
    modifies (visible object(s) in parameter table);
    ensures φ_post;
    }
```

Figure 12: Template for Larch/C++ transition interface specification

The shared tier of Larch is *Property-oriented specification method* and Larch/C++ uses the models from the shared tier(LSL). In LSL, objects are built from types, and operations on types are given as assertions in first-order predicate logic. For this reason, the validation and verification of implementation from TROM design model, can use Larch/C++ as a facility.

6.2 C++ Code Generation

In the implementation, the C++ code of reactive objects in the system is generated from their class specifications in TROM formalism.

According to the implementation model, a *generic reactive class*, is a derived class of Class TROM. In implementation, the C++ code of each *generic reactive class* will be generated automatically. Each generated *generic reactive class*, which with the properties inherited from Class TROM, will be initialized when it is created. Here the properties initialized are *events*, *states*, *attributes*, *transitions* and *constraints*. Other properties, such as *ports*, will be initialized in subsystem configuration class (SCS)

The property *events* is a set of events, each event is an instance of class Event. The instantiation of the each Event object is through the constructor function of Class Event . That is,

```
events = set ⟨events⟩();
events.add(event(event/1s name, event1's type, event1's porttype));
events.add(event(event2's name, event2's type, event2's porttype));
.
.
.
```

The instantiation of each state object and the initialization of property *states* in the class, are the same as *events*. However, because of the hierarchical structure of state, it is more difficult to be implemented.

The property *transitions* refers to a set of transition_specifications, where each transition_specification is defined as a *generic transition class* (GTC). Each *generic transition class* (GTC) is a derived class of Class Transition. With a group of virtual functions (predicates) that should be indicated, the Class Transition is a polymorphic type, its derived class must define the virtual functions inherited from Class Transition.

The Constraints is a set of time_constraints, the initialization is the same as Events and States. This property involves the time calculation during the computation.

The data structure *event_buffer* is used for communications, and the *assignment vector* as well as *reaction vector* are maintained during the system execution. Both of the two vectors are specific to individual reactive object, and are of thread-specific context.

A subsystem specifies a complete and consistent view of a solution to one aspect of the problem being modeled. Each subsystem encapsulates the collaboration, and concurrent interactions among a collection of objects is instantiated from the second tier class specifications. A *system configuration specification* (SCS) specifies a reactive system (or a subsystem) by composing of objects or smaller subsystems.

> **SCS** ⟨ identifier ⟩
> **Include:**
> **Instantiate:**
> **Configure:**
> **Constraints:**
> **end**

The template above gives the syntax for system (or subsystem) configuration specifications. The keyword **SCS** introduces the identifier for the subsystem. The **Include** clause is optional and is useful for importing system (subsystem) definitions from other system configuration specification (SCS). A reactive object is defined in the **Instantiate** clause by parameterizing a TROM with cardinality of ports and the values of the active attributes (if any) in the initial state of the TROM. The **Configure** clause defines a configuration obtained by composing objects specified in the **Instantiate** clause and the subsystem specifications imported through the **Include** clause. The symbol $\leftrightarrow$ is used here to note communication links between compatible ports of interacting objects. For example, a link controller1 @G1 $\leftrightarrow$ gate1.@S1 in the **Configure** clause connects the port G1 of object *controller1* and port S1 of object *gate1*. The **Constraints** clause specifies the constraints among the attribute initialization of a collection of interacting objects.

6.3 Conformance

The conformance analysis focuses on the *transition_specifications* in TROM, it is because that the behaviors of TROM is governed by the predicates in transition specifications.

According to the operational semantics of TROM, a computational step has an associated transition and it enables a step of computation when the values of the attributes in the source state of the transition satisfy the enabling condition of the transition. The *synchronization* condition for computation (section 2.1) means that the port_condition is satisfied for the message passing. In the destination state, the new values of the active attributes should satisfy the post_condition of the transition. A post_condition specified as *true* means that the active attributes are unconstrained; Whilst a post_condition specified as *false*, from the point of view of designer, means that the transition specification is inconsistency.

In Larch/C++, the precondition to invoke the transition, is described in the **requires** clause; and the postcondition that the transition establishes upon termination, is in the **ensures** clause. The precondition of the transition must logically imply the postcondition.

Two data structures, the *assignment vector* and *reactive vector*, are handled during the execution of my implementation. By knowing them, we can check whether the conditions of transition specification are satisfied or not.

The data structure *assignment vector* is obtained when code is generated from TROM class specification. The attributes are specified in **Attributes** section of the TROM specification, while its data type is given in **Traits** section.

Consider the following TROM specification sections,

⋮
Attributes:inSet:PSet
Traits:Set[@P,PSet]
⋮

The data structure *inSet* is defined as an *attribute* with the type *PSet*. The *PSet* is a *Trait*. In C++ implementation, *PSet* is a user defined type.

An *assertion* in C++, is simply a statement that holds a given logical criterion. Assertions can be used to express preconditions and postconditions of a function. That is, checking the basic assumptions about input and verifying whether the function leaves the world in the excepted state upon exit.

The assertions associated with the computation, are given by the predicates of the specified transitions in the specific case.

As an example, consider the following transition interface specification in Larch/C++. It is from the rail-road crossing problem.

```
R4(obj pid,obj inSet)
{
trigger Exit?
requires member(pid,inSet); size(inSet)=1; curstate="monitor";
modifies inSet;
ensures inset'= delete(pid,inSet);
}
```

The informal semantics of this transition is : when event *Exit* occurs in state *monitor*, if the identified **pid** of the port where event *Exit* was received is a member of *inSet* and if the size of *inSet* is equal to 1, then this is the only train in the crossing, and the current **pid** is deleted from *inSet* and the object goes into state *deactivate*.

In the execution of the resulting program, if the event *Exit?* as trigger is received, the current state is *monitor*, the assertion *inSet.member(pid)* as port_condition is satisfied, and *inSet.size*() = 1 as enabling condition is also satisfied, then the transition will be invoked.

After the transition, the assertion *inSet′* = *inSet.delete(pid)* should be satisfied. If not, the execution will fail with an *exception*. An *exception* is provided to help deal with error reporting, that is:

```
post_conform(vector assignment_vactor)
{
if (postcondition == false)
throw Post_error(assignment_vector);
return (assignment_vector);
}
```

The function *post_conform*() either returns the vector *assignment_vector* or throws a *Post_error*. It is to say that when the *post_condition* is not satisfied, the execution will "throw" the problem to special module for dealing with the exception. The function to handle the problem will *catch* this *exception* and report the error.

When the system executes the program segment corresponding to a transition specification, the conformance is processed to check whether the generated C++ code of a transition meets the requirements in its Larch/C++ specification. If the status of the running case satisfies the predicates in the **requires** clause before a transition, and satisfies the predicates in the **ensures** clause after the transition, then the implementation of this transition is guaranteed to be consistent with its Larch/C++ interface specification.

The process of conformance goes with the case running phase in which some abstract test cases are selected. These cases are used to test the execution of the implementation. By ensuring the consistency between the implementation and its specification, the implementation is guaranteed to be what it was intended and designed to do.

Reactive objects are event-driven, that makes it possible to describe them in terms of their dynamic behaviors. As we discussed in previous section, the specification of a reactive object is succinctly captured by a set of behaviors, and each behavior is a trace of massage passing that occurs at the interface between the object and the environment. The *event sequence*, which generated by each reactive object of the system during the execution, is used as the test case for the conformance. By this approach, we can check if the implemented system is running in the way described in the specification.

A TROM's behavior, which associated with a set of transitions, is marked by sequences of events appearing at its incoming ports (*incoming events*) and the responses which in the form of *outgoing events*. From the point of view of conformance, the purpose of test is to check if the implementation satisfies its specification requirements with respect to a transition. A test case does this by applying sequences of events to the implementation and inspecting the status of the reactive object with the predicates stated in the Larch/C++ specification. For each testing process, a test result will be reported to indicate whether the implementation passes the test, or fails (or behaves in an inconclusive manner). If the observed behavior of the implementation is allowed by the Larch/C++ specification and the test purpose is satisfied, then a "pass" result is posted and the execution will be continued. If any run-time status of the system is not consistent with the Larch/C++ specification, then the test case gives a "fail" result and reports the error. When an inconsistency is reported, the history of the computation can be examined to find the errors that might have caused such an inconsistency. The iterative design, validate, and implement cycle in TROMLAB framework promotes re-implementation with only necessary changes.

Chapter 7

Case Study

7.1 Train-Gate-Controller Problem

The *Train-gate-controller problem* has been discussed previously in [HL94], [Ach95] and [AM98] as a case study to illustrate the expressive of the TROM formalism. In this chapter we use it to show how the C++ code is generated from its design model.

In *Train-gate-controller* problem, several trains cross a gate independently and simultaneously using non-overlapping tracks. A train may choose to cross any gate on its way. A controller controls each gate. When a train is near the gate, it sends a message to the associated controller which commands the gate to close. While While the train exits the crossing, it sends a message to the controller which instructs the gate to open.

To ensure safety of the system, some time constraints are attached on messages. For example, a train should be inside the crossing within 2 to 4 time units after sending the message indicating that it is near the gate. The train should send a message indicating that it is ready to exit the crossing within 6 time units from the first message. Within 1 time unit after receiving the initial message from the train, the controller should let the gate lower and the controller begins to monitor it. If the controller receives a *near* message when it is monitoring the gate, it should keep on monitoring the gate. The controller will let the gate raise within 1 time unit after the last train exits the crossing. The gate must close within 1 time unit after the

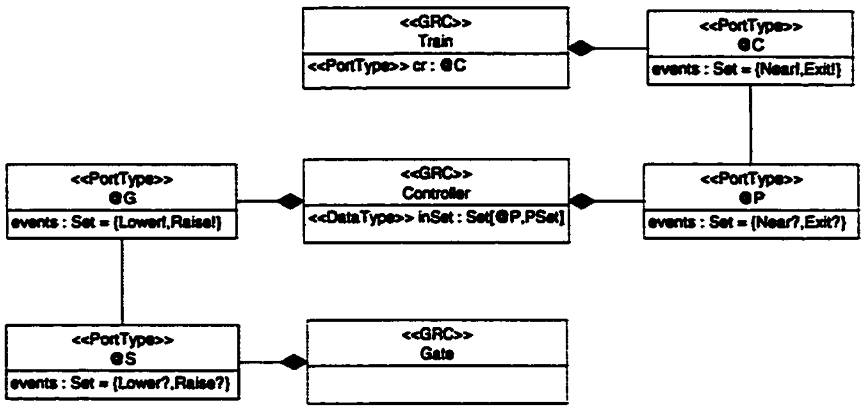

<<GRC>>
Train
<<PortType>> cr : @C
<<PortType>>
@C
events : Set = {Near!,Exit!}
<<PortType>>
@G
events : Set = {Lower!,Raise!}
<<GRC>>
Controller
<<DataType>> inSet : Set[@P,PSet]
<<PortType>>
@P
events : Set = {Near?,Exit?}
<<PortType>>
@S
events : Set = {Lower?,Raise?}
<<GRC>>
Gate

Figure 13: Main Class Diagram for Train-Gate-Controller

controller let it lower. The gate must open within the period of 1 to 2 time unit after the controller instructs it to rise.

The class diagram, as depicted in Figure13, shows the three generic reactive classes: Train, Gate, Controller, and their connections.

Train has a group of ports with port type @C. Controller has a group of of ports with port types @G and @P.Gate has a group of ports with port type @S.

There is an association between port type @C of Train and port type @P of Controller, which means that the generic reactive class Train uses port(s) with port type @C to communicate to the generic reactive class Controller through the port(s) with port type @P. The association between port type @S of Gate and port type @G of controller, means that generic reactive class Controller uses port(s) with port type @G to communicate to generic reactive class Gate through the port(s) with port type @S.

Train has one port type @C. At the port(s) with this port type, the outgoing event *Near!* and the incoming event *Exit?* may occur. Train has one attribute *cr*, whose type is port type @C.

Controller has two port type: @P and @G. The incoming event *Near?* and incoming event *Exit?* may occur at port(s) with port type @P. The outgoing event *Lower!* and outgoing event *Raise!* may occur at port(s) with port type @G. Controller has one attribute *inSet*, its type is the abstract data type defined in the LSL trait

Set[@P,PSet].

Gate has one port type @S. The incoming event *Lower?* and incoming event *Raise?* may occur at port with port type @C.

Figure 14 and Figure 15 show the statechart diagrams and the formal specification of generic reactive class Train .

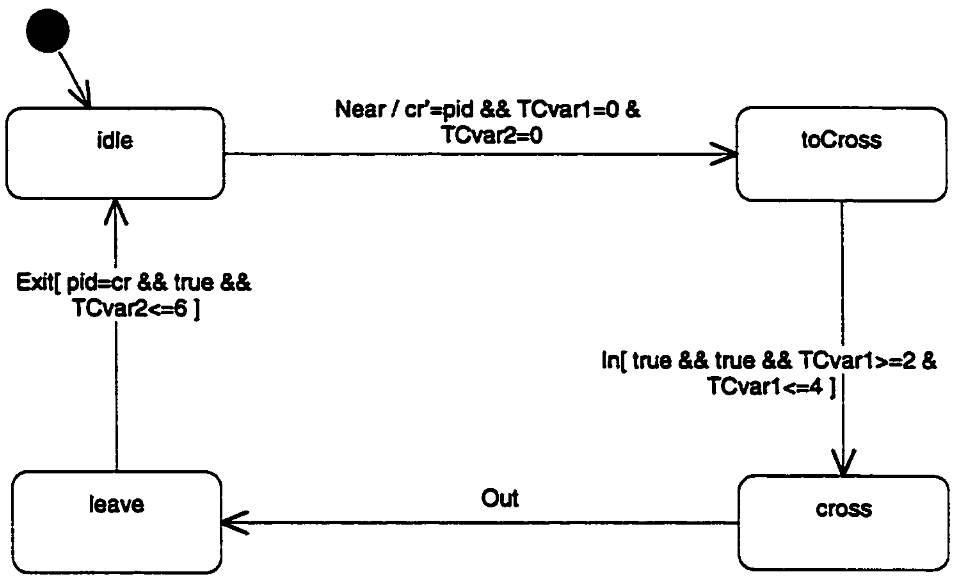

idle
Near / cr'=pid && TCvar1=0 & TCvar2=0
toCross
Exit[pid=cr && true && TCvar2<=6]
In[true && true && TCvar1>=2 & TCvar1<=4]
leave
Out
cross

Figure 14: Statechart Diagram for Train

A Train object can be in one of four states: *idle*, *toCross*, *cross*, *leave*. Here, *idle* is the initial state.

When event *Near!* occurs in state *idle*, attribute *cr* is set to **pid**. The **pid** is the identifier of the port at which event *Near!* occurs. A possible transition here is a constraining transition for two time constraints, *TCvar1: R1, In, [2, 4]*, {} and *TCvar2: R1, Exit, [0, 6]*, {} . By the transition, Train goes into state *toCross*.

A transition from state *toCross* to state *cross* happens when internal event *In* occurs in state *toCross*, and if the time constraint condition *TCvar1* ≥ 2 AND *TCvar1* ≤ 4 is true. This time constraint means that internal event *In* should occur within 2 to 4

```
Class Train [@C]
Events: Near!@C, Out, Exit!@C, 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 15: Formal specification for GRC Train

time units after *Near!* occurs in state *idle*.

When internal *Out* occurs in state *cross*, Train goes into state *leave*.

A transition from state *leave* to *idle* happens when event *Exit!* occurs in state *leave*, if the attribute *cr* has the value **pid** (the identifier of the port where *Exit* occurs), and if the time constraints condition *TCvar2* ≤ 6 is true. This time constraint means that event *Exit* should occur within 6 time units after event *Near* occurs in state *idle*.

Figure 16 and Figure 17 show the statechart diagram and the formal specification of generic reactive class Controller

A Controller object can be in one of four states: *idle*, *activate*, *monitor*, and *deactivate*. *idle* is the initial state.

When event *Near?* occurs in state *idle*, the attribute *inSet* is modified to include the new entry **pid** (the identifier of port where *Near?* occurs). The Controller goes into state *activate*. This transition is the constraining transition of time constraint

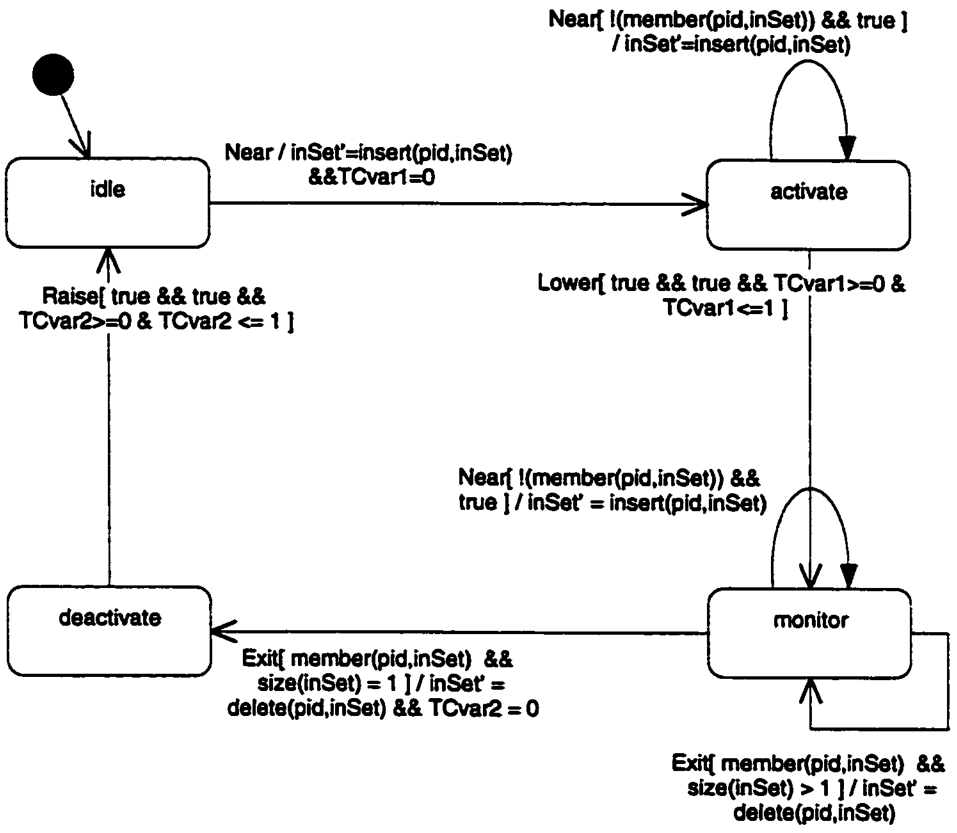

Near[!(member(pid,inSet)) && true] / inSet'=insert(pid,inSet)
idle
Near / inSet'=insert(pid,inSet) &&TCvar1=0
activate
Raise[true && true && TCvar2>=0 & TCvar2 <= 1]
Lower[true && true && TCvar1>=0 & TCvar1<=1]
Near[!(member(pid,inSet)) && true] / inSet' = insert(pid,inSet)
deactivate
monitor
Exit[member(pid,inSet) && size(inSet) = 1] / inSet' = delete(pid,inSet) && TCvar2 = 0
Exit[member(pid,inSet) && size(inSet) > 1] / inSet' = delete(pid,inSet)

Figure 16: Statechart Diagram for Controller

```
Class Controller [@P, @G]
Events: Lower!@G, Near?@P, Raise!@G, Exit?@P
States: *idle, activate, deactivate, monitor
Attributes: inSet:PSet
Traits: Set[@P,PSet]
Attribute-Function: activate → {inSet}; deactivate → {inSet}; monitor → {inSet};
          idle → {};
Transition-Specifications:
     R1: <activate,monitor>; Lower(true);
          true ⟹ true;
     R2: <activate,activate>; Near(!(member(pid,inSet)));
          true ⟹ inSet′=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 17: Formal specification for GRC Controller

TCvar1: R7, Lower, [0, 1], {}.

When event *Near?* occurs in state *activate* from a different Train (**pid** is not already a member of set *inSet*), the attribute *inSet* is modified to include the new **pid** (identifier of the port where the new event *Near?* occurs). The Controller remains in the state *activate*.

When event *Lower!* occurs in state *activate*, if the time constraint condition $TCvar1 \leq 1$ is true, the Controller goes into state *monitor*. This time constraint means that event *Lower!* should occur within one time unit after event *Near?* occurs in state *idle*.

When event *Near?* occurs in state *monitor* from a different Train(**pid** is not already a member of set *inSet*), the attribute *inSet* is modified to include the new **pid**(identifier of the port where the new event *Near?* occurs). The Controller remains in state *monitor*.

When event *Exit?* occurs in state *monitor*, if the identifier **pid** of the port where event *Exit?* was received is a member of *inSet* and if the size of *inSet* is greater than 1, then the current **pid** is deleted from *inSet* and Controller remains in state *monitor*.

When event *Exit?* occurs in state *monitor*, if the identifier **pid** of the port where event *Exit?* was received is a member of *inSet* and if the size of *inSet* equals to 1, then the current **pid** is deleted from *inSet* and Controller goes into state *deactivate*. This is the constraining transition for time constraint *TCvar2: R4, Raise, [0, 1],* {}.

When event *Raise!* occurs in state *deactivate*, if time constraint condition $TCvar2 \leq 1$ is true, the Controller goes into state *idle*. This time constraint means that event *Raise!* should occur within 1 time unit after event *Exit?* was received from the last Train in the crossing.

Figure 18 and Figure 19 show the statechart diagram and the formal specification of generic reactive class Gate .

A Gate object can be in one of four states: *opened*, *toClose*, *closed*, and *toOpen*.

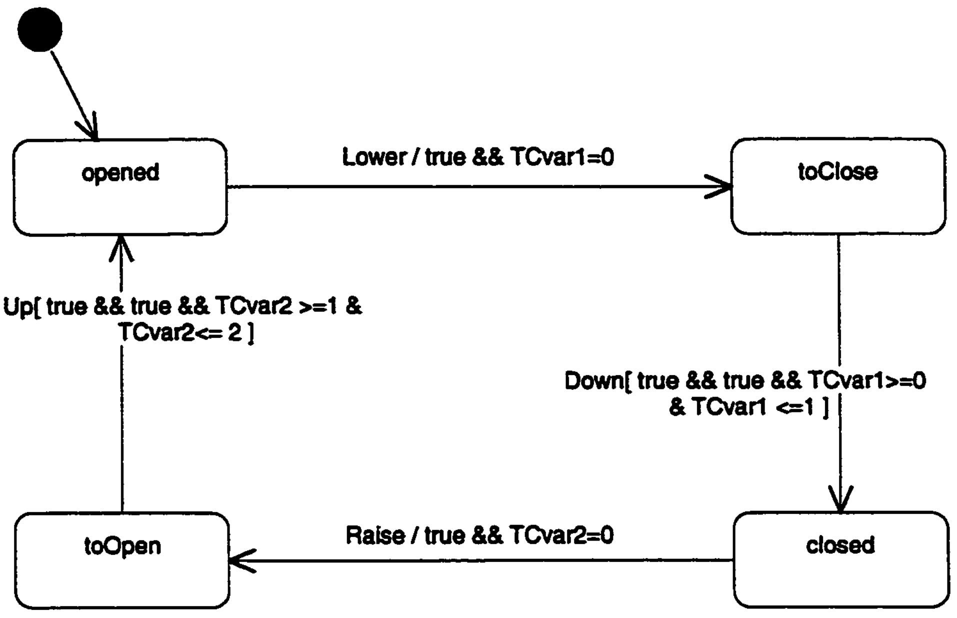

opened
Lower / true && TCvar1=0
toClose
Up[true && true && TCvar2 >=1 & TCvar2<= 2]
Down[true && true && TCvar1>=0 & TCvar1 <=1]
toOpen
Raise / true && TCvar2=0
closed

Figure 18: Statechart Diagram for Gate

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 $\Longrightarrow$ true;
R2: <toClose,closed>; Down(true); true $\Longrightarrow$ true;
R3: <toOpen,opened>; Up(true); true $\Longrightarrow$ true;
R4: <closed,toOpen>; Raise(true); true $\Longrightarrow$ true;

Time-Constraints:

TCvar1: R1, Down, [0, 1], {};
TCvar2: R4, Up, [1, 2], {};

end

Figure 19: Formal specification for GRC Gate

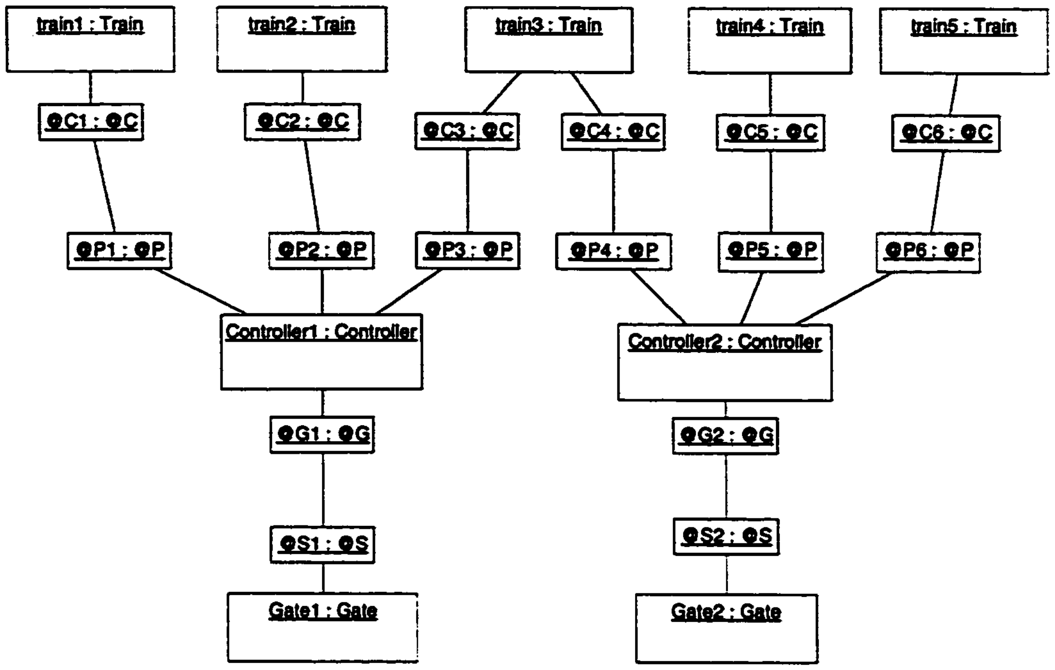


Figure 20: Collaboration diagram for subsystem TrainGateController2

opened is the initial state.

When event *Lower?* occurs in state *opened*, the Gate goes into state *toClose*. This is the constraining transition of time constraint *TCvar1: R1, Down, [0, 1]*, {}.

When event *Down* occurs in state *toClose*, if *TCvar1* ≤ 1 is true, meaning that event *Down* should occurs within 1 time unit after event *Lower* occurs in state *opened*, Gate goes into state *closed*.

When event *Raise?* occurs in state *closed*, the Gate goes into state *toOpen*. This is the constraining transition of time constraint *TCvar2: R4, Up, [1, 2]*, {}.

When event *Up* occurs in state *toOpen*, if $TCvar2 \geq 1$ and $TCvar1 \leq 2$ is true, meaning that event *Up* should occurs within 1 to 2 time unit after event *Raise?* occurs in state *closd*, Gate goes into state *opened*.

The Railroad subsystem with 5 trains , 2 Controllers and 2 Gates, is an example used here for code generation. Figure 20 and Figure 21 show the Collaboration diagram and the formal specification of the subsystem.

```
SCS TrainGateController2
      Includes:
      Instantiate:
            Gate2::Gate[@S:1];
            Gate1::Gate[@S:1];
            Controller1::Controller[@P:3, @G:1];
            Controller2::Controller[@P:3, @G:1];
            train1::Train[@C:1];
            train2::Train[@C:1];
            train3::Train[@C:2];
            train4::Train[@C:1];
            train5::Train[@C:1];
      Configure:
            Gate1.@S1:@S ↔ Controller1.@G1:@G;
            Controller2.@G2:@G ↔ Gate2.@S2:@S;
            Controller1.@P2:@P ↔ train2.@C2:@C;
            Controller1.@P1:@P ↔ train1.@C1:@C;
            Controller1.@P3:@P ↔ train3.@C3:@C;
            Controller2.@P5:@P ↔ train4.@C5:@C;
            Controller2.@P6:@P ↔ train5.@C6:@C;
            Controller2.@P4:@P ↔ train3.@C4:@C;
end
```

Figure 21: Formal specification for subsystem TrainGateController2

7.2 Implementation

The generated Class Train is an inheritance of Class TROM. As shown in Figure 22, the constructor function *Train()* initializes the properties *events*, *states*, *attributes*, *transitions* and *constraints*, when the Class Train is created.

Each transition_specification in design model, is generated as a *generic transition class* (GTC) in C++. Each GTC is a derived class of Class Transition which with polymorphic type. So the virtual functions inherited from Class Transition, such as *enabling_condition*, *port_condition*, *post_condition*, and *effect()* are defined the details in each GTC.

The *events* in Train is initialized as a set of events, such that *Near!@C*, *Out*, *Exit!@C* and *In*.

The *states* in Train is initialized as a set of states: *idle*, *cross*, *leave* and *toCross*. The state *idle* is the initial state. And all of the four states are without substates in this design model.

The *attributes* in Train is initialized as a set of attributes with only one element: *cr*, *cr* is a attribute with the type of *@C*.

The *constraints* in Train is initialized as a set of time_constraints : *TCvar1* and *TCvar2*. The transition_specification of *TCvar1* is TrainR1, the constrained event of *TCvar1* is *In*, and the response time bounds is [2,4], the set of disable states is null.

In *TCvar2*, The transition_specification is TrainR1, the constrained event is *Exit*, the response time bounds is [2,4], and the set of disable states is null.

The *transitions* in Train is initialized as a set of *generic transition classes*(GTCs), such that Class TrainR1, Class TrainR2, Class TrainR3 and Class TrainR4.

The transition specifications in design model, are translated into Larch/C++ interface specification at first. In these Larch/C++ Modules, the precondition, postcondition of transition_specifications in Train, together with the visible object modified in it, is specified. Figure 23 shows the generated Larch/C++ interface specification of transition TrainR1. In this generated interface specification, the pre-condition of transition TrainR1 is *current state equals "idle"*, (the condition is specified in **requires** clause).

```
class Train: public TROM
{
     public:
          Train()
          {
          events = set ⟨ event ⟩();
          events.add(event("Near", "!", "@C"));
          events.add(event("Out", "", ""));
          events.add(event("Exit", "!", "@C"));
          events.add(event("In", "", ""));
          states = set ⟨state⟩();
          states.add(state("idle", true, set ⟨ state ⟩()));
          states.add(state("cross", false, set ⟨ state ⟩()));
          states.add(state("leave", false, set ⟨ state ⟩()));
          states.add(state("toCross", false, set ⟨ state ⟩()));
          atts = set ⟨attribute⟩ ();
          atts.add(attribute("cr",true,"","@C"));
          trans = set ⟨ transition ⟩();
          trans.add(TrainR1());
          trans.add(TrainR2());
          trans.add(TrainR3());
          trans.add(TrainR4());
          cons = set⟨constraint⟩();
          cons.add(constraint("TCvar2", "R1", "Exit", 0, 6, set ⟨ string ⟩()));
          cons.add(constraint("TCvar1", "R1", "In", 2, 4, set ⟨ string ⟩()));
          }
}
```

Figure 22: Generated C++ Code of Class Train

The post-condition is *the updated value of cr equals* **pid**. (specified in **ensures** clause). The visible object modified in this function is *cr*.

The generated Class TrainR1 is a generic transition class. In Class TrainR1, the virtual functions inherited from Class Transition are redefined.

Figure 24 shows the generated C++ code of Class TrainR1. The *name* of the transition, the source state *sstate*, the destination state *dstate* and the triggering event *eventname* are initialized in constractor function *TrainR1()*. The virtual functions *port_condition()*, *enabling_condition()*, *effect()* and *post_condition()* are defined the

```
TrainR1(obj cr)
{
requires curstate="idle";
modifies cr;
ensures cr'=pid;
}
```

Figure 23: Generated Larch/C++ interface specification of TrainR1

functions' details in this specific transition.

Figure 25 is the generated Larch/C++ interface specification of transition TrainR2. Figure 26 shows the generated code of this transition.

Figure 27 is the generated Larch/C++ interface specification of transition TrainR3, and Figure 28 shows its generated code in C++.

Figure 29 is the generated Larch/C++ interface specification of transition TrainR4. Figure 30 is the generated code of Class TrainR4.

The generic reactive class Controller is also a derived class of Class TROM, The properties in Controller are initialized by its constructor function *Controller()* in the same way as being done in Class Train.

As well as shown in Class Train, each transition specification in Controller also refers to a generic transition class in which the virtual functions are defined as well.

Figure 31 shows the generated code of Class Controller.

The *constraints* in Controller is initialized as a set of time_constraints : *TCvar1* and *TCvar2*. The transition_specification of *TCvar1* is ControllerR7, the constrained event of *TCvar1* is *Lower*, and the response time bounds is [0,1], the set of disable states is null. In *TCvar2*, the transition_specification is ControllerR4, the constrained event is *Raise*, the response time bounds is [0,1], and the set of disable states is null.

```
class TrainR1 : public transition
{
private:
     string cr, cr/;
public:
     TrainR1()
     {
     name="TrainR1";
     sstate="idle";
     dstate="toCross";
     eventname="Near"
     };
     bool port-condition(string pid)
     { return true};
     bool enabling-condition(string pid)
     { return true};
     void effect()
     { AssignmentVector(cr/,pid);};
     bool post-condition(string pid)
     {return cr/==pid}
}
```

Figure 24: Generated C++ Code of Class TrainR1

The *transitions* in Controller is initialized as set of generic transition classes: ControllerR1, ControllerR2, ControllerR3, ControllerR4, ControllerR5, ControllerR6 and ControllerR7.

Figure 32 is the generated Larch/C++ interface specification of transition ControllerR1, and Figure 33 shows the Class ControllerR1 which is generated from the

```
TrainR2()
{
requires curstate="cross";
}
```

Figure 25: Generated Larch/C++ interface specification of TrainR2

```
class TrainR2 : public transition
{
      private:
            string cr,cr';
      public:
            TrainR2()
            {
            name="TrainR2";
            sstate="cross";
            dstate="leave";
            eventname="Out";
            };
            bool port-condition(string pid)
            { return true};
            bool enabling-condition(string pid)
            { return true};
            void effect()
            {};
            bool post-condition(string pid)
            {return true}
}
```

Figure 26: Generated C++ Code of Class TrainR2

```
TrainR3()
{
requires pid=cr; curstate="leave";
}
```

Figure 27: Generated Larch/C++ interface specification of TrainR3

```
class TrainR3 : public transition
{
      private:
            string cr,cr/;
      public:
            TrainR3()
            {
            name="TrainR3";
            sstate="leave";
            dstate="idle";
            eventname="Exit";
            };
            bool port-condition(string pid)
            { return pid==cr};
            bool enabling-condition(string pid)
            { return true};
            void effect()
            {};
            bool post-condition(string pid)
            {return true}
}
```

Figure 28: Generated C++ Code of Class TrainR3

```
TrainR4()
{
requires curstate="toCross";
}
```

Figure 29: Generated Larch/C++ interface specification of TrainR4

```
class TrainR4 : public transition
{
      private:
            string cr,cr/;
      public:
            TrainR4()
            {
            name="TrainR4";
            sstate="toCross";
            dstate="cross";
            eventname="In"
            };
            bool port-condition(string pid)
            { return true};
            bool enabling-condition(string pid)
            { return true};
            void effect()
            {};
            bool post-condition(string pid)
            {return true}
}
```

Figure 30: Generated C++ Code of Class TrainR4

Larch/C++ interface specification.

Figure 34 is the generated Larch/C++ interface specification of transition ControllerR2, and Figure 35 is the generated C++ code of Class ControllerR2.

Figure 36 is the generated Larch/C++ interface specification of transition ControllerR3. Figure 37 is the generated C++ code of ControllerR3

Figure 38 and Figure 39 are the generated Larch/C++ interface specification of transition ControllerR4 and the C++ code generated form this Larch/C++ specification.

Figure 40 is the Larch/C++ interface specification of transition ControllerR5, Figure 41 is the generated C++ code of this transition.

Figure 42 is the generated Larch/C++ interface specification of transition ControllerR6, Figure 43 is the generated C++ code of Class ControllerR6.

Figure 44 is the generated Larch/C++ interface specification of transition ControllerR7, Figure 45 is the generated C++ code.

Figure 46 shows the generated code of Class *Gate*. The *constraints* in Gate is a set of time_constraints : *TCvar1* and *TCvar2*. The transition_specification of *TCvar1* is GateR1, the constrained event of *TCvar1* is *Down*, and the response time bounds is [0,1], the set of disable states is null. In *TCvar2*, The transition_specification is GateR4, the constrained event is *Up*, the response time bounds is [1,2], and the set of disable states is null.

The *transitions* in Gate is initialized as a set of *generic transition classes*: GateR1,

```
class Controller: public TROM
    {
    public:
        Controller()
        {
        events = set ⟨ event ⟩();
        events.add(event("Lower", "!", "@G"));
        events.add(event("Near", "?", "@P"));
        events.add(event("Raise", "!", "@G"));
        events.add(event("Exit", "?", "@P"));
        states = set ⟨state⟩ ();
        states.add(state("idle", true, set⟨state⟩()));
        states.add(state("activate", false, set⟨state⟩()));
        states.add(state("deactivate", false, set⟨state⟩()));
        states.add(state("monitor", false, set⟨state⟩()));
        atts = set⟨attribute⟩();
        atts.add(attribute("inSet", false, "PSet",""));
        trans = set⟨transition⟩();
        trans.add(ControllerR1());
        trans.add(ControllerR2());
        trans.add(ControllerR3());
        trans.add(ControllerR4());
        trans.add(ControllerR5());
        trans.add(ControllerR6());
        trans.add(ControllerR7());
        cons = set⟨constraint⟩();
        cons.add(constraint("TCvar1", "R7", "Lower", 0, 1, set⟨string⟩()));
        cons.add(constraint("TCvar2", "R4", "Raise", 0, 1, set⟨string⟩()));
        }
    }
```

Figure 31: Generated C++ Code of Class Controller

```
ControllerR1()
{
requires curstate="activate";
}
```

Figure 32: Generated Larch/C++ interface specification of ControllerR1

```
class ControllerR1 : public transition
{
        private:
                PSet inSet,inSet/;
        public:
                ControllerR1()
                {
                name="ControllerR1";
                sstate="activate";
                dstate="monitor";
                eventname="Lower";
                };
                bool port-condition(string pid)
                { return true};
                bool enabling-condition(string pid)
                { return true};
                void effect()
                {};
                bool post-condition(string pid)
                {return true}
}
```

Figure 33: Generated C++ Code of Class ControllerR1

```
ControllerR2(obj inSet,obj pid)
{
requires !(member(pid, inSet)); curstate="activate";
modifies inSet;
ensures inSet/=insert(pid,inSet);
}
```

Figure 34: Generated Larch/C++ interface specification of ControllerR2

```
class ControllerR2 : public transition
{
      private:
            PSet inSet,inSet′;
      public:
            ControllerR2()
            {
            name="ControllerR2";
            sstate="activate";
            dstate="activate";
            eventname="Near";
            };
            bool port-condition(string pid)
            { return !(inSet.member(pid))};
            bool enabling-condition(string pid)
            { return true};
            void effect()
            { AssignmentVector(inSet′, inSet.insert(pid)); };
            bool post-condition(string pid)
            {return inSet′==inSet.insert(pid)
            }
}
```

Figure 35: Generated C++ Code of Class ControllerR2

```
ControllerR3()
{
requires curstate="deactivate";
}
```

Figure 36: Generated Larch/C++ interface specification of ControllerR3

```
class ControllerR3 : public transition
{
      private:
            PSet inSet,inSet';
      public:
            ControllerR3()
            {
            name="ControllerR3";
            sstate="deactivate";
            dstate="idle";
            eventname="Raise"
            };
            bool port-condition(string pid)
            { return true};
            bool enabling-condition(string pid)
            { return true};
            void effect()
            {};
            bool post-condition(string pid)
            {return true}
}
```

Figure 37: Generated C++ Code of Class ControllerR3

```
ControllerR4(obj pid,obj inSet)
{
requires member(pid,inSet); curstate="monitor";
modifies inSet;
ensures inset'=delete(pid,inSet);
}
```

Figure 38: Generated Larch/C++ interface specification of ControllerR4

```
class ControllerR4 : public transition
{
      private:
            PSet inSet,inSet';
      public:
            ControllerR4()
            {
            name="ControllerR4";
            sstate="monitor";
            dstate="deactivate";
            eventname="Exit";
            };
            bool port-condition(string pid)
            { return inSet.member(pid)};
            bool enabling-condition(string pid)
            { return inSet.size()==1};
            void effect()
            { AssignmentVector(inSet',inSet.delete1(pid)); };
            bool post-condition(string pid)
            {return inSet'==inSet.delete1(pid)}
}
```

Figure 39: Generated C++ Code of Class ControllerR4

```
ControllerR5(obj pid,obj inSet)
{
requires member(pid,inSet); curstate="monitor";
modifies inSet;
ensures inse't=delete(pid,inSet);
}
```

Figure 40: Generated Larch/C++ interface specification of ControllerR5

```
class ControllerR5 : public transition
{
      private:
            PSet inSet,inSet′;
      public:
            ControllerR5()
            {
            name="ControllerR5";
            sstate="monitor";
            dstate="monitor";
            eventname="Exit";
            };
            bool port-condition(string pid)
            { return inSet.member(pid)};
            bool enabling-condition(string pid)
            { return inSet.size() > 1};
            void effect()
            { AssignmentVector(inSet′,inSet.delete1(pid)); };
            bool post-condition(string pid)
            {return inSet′==inSet.delete1(pid)}
}
```

Figure 41: Generated C++ Code of Class ControllerR5

```
ControllerR6(obj inSet,obj pid)
{
requires !(member(pid,inSet)); curstate="monitor";
modifies inSet;
ensures inSet′=insert(pid,inSet);
}
```

Figure 42: Generated Larch/C++ interface specification of ControllerR6

```
class ControllerR6 : public transition
{
      private:
            PSet inSet,inSet′;
      public:
            ControllerR6()
            {
            name="ControllerR6";
            sstate="monitor";
            dstate="monitor";
            eventname="Near";
            };
            bool port-condition(string pid)
            { return !(inSet.member(pid))};
            bool enabling-condition(string pid)
            { return true};
            void effect()
            { AssignmentVector(inSet′,inSet.insert(pid)); };
            bool post-condition(string pid)
            {return inSet′==inSet.insert(pid)}
};
```

Figure 43: Generated C++ Code of Class ControllerR6

```
ControllerR7(obj inSet,obj pid)
{
requires curstate="idle";
modifies inSet;
ensures inSet′=insert(pid,inSet);
}
```

Figure 44: Generated Larch/C++ interface specification of ControllerR7

```
class ControllerR7 : public transition
{
      private:
            PSet inSet,inSe/t;
public:
            ControllerR7()
            {
            name="ControllerR7";
            sstate="idle";
            dstate="activate";
            eventname="Near";
            };
            bool port-condition(string pid)
            { return true};
            bool enabling-condition(string pid)
            { return true};
            void effect()
            { AssignmentVector(inSet/,inSet.insert(pid)); };
            bool post-condition(string pid)
            {return inSet/==inSet.insert(pid)}
}
```

Figure 45: Generated C++ Code of Class ControllerR7

```
class Gate: public TROM {
      public:
            Gate() {
            events = set⟨event⟩();
            events.add(event("Lower", "?", "@S"));
            events.add(event("Down", "", ""));
            events.add(event("Up", "", ""));
            events.add(event("Raise", "?", "@S"));
            states = set⟨state⟩();
            states.add(state("opened", true, set⟨state⟩()));
            states.add(state("toClose", false, set⟨state⟩()));
            states.add(state("toOpen", false, set⟨state⟩()));
            states.add(state("closed", false, set⟨state⟩()));
            atts = set⟨attribute⟩();
            trans = set⟨transition⟩ ();
            trans.add(GateR1());
            trans.add(GateR2());
            trans.add(GateR3());
            trans.add(GateR4());
            cons = set⟨constraint⟩();
            cons.add(constraint("TCvar1", "R1", "Down", 0, 1, set⟨string⟩()));
            cons.add(constraint("TCvar2", "R4", "Up", 1, 2, set⟨state⟩()));
            }
      }
```

Figure 46: Generated C++ Code of Class Gate

GateR2, GateR3 and GateR4.

The generated Larch/C++ interface specification of transition GateR1, and the generated C++ code of Class GateR1 are shown in Figure 47 and Figure 48.

Figure 49 and Figure 50 are the generated Larch/C++ interface specification of transition GateR2, and the generated C++ code of Class GateR2.

Figure 51 and Figure 52 are the generated Larch/C++ interface specification of transition GateR3 and the generated code of Class GateR3.

Figure 53 and Figure 54 are the generated Larch/C++ interface specification of transition GateR4 and the generated C++ code of Class GateR4.

```
GateR1()
{
requires curstate="opened";
}
```

Figure 47: Generated Larch/C++ interface specification of GateR1

```
class GateR1 : public transition
{
      private:

      public:
      GateR1()
      {
      name="GateR1";
      sstate="opened";
      dstate="toClose";
      eventname="Lower";
      };
      bool port-condition(string pid)
      { return true};
      bool enabling-condition(string pid)
      { return true};
      void effect()
      {};
      bool post-condition(string pid)
      {return true}
}
```

Figure 48: Generated C++ Code of Class GateR1

```
GateR2()
{
requires curstate="toClose";
}
```

Figure 49: Generated Larch/C++ interface specification of GateR2

```
class GateR2 : public transition
{
    private:

    public:
        GateR2()
        {
        name="GateR2";
        sstate="toClose";
        dstate="closed";
        eventname="Down";
        };
        bool port-condition(string pid)
        { return true};
        bool enabling-condition(string pid)
        { return true};
        void effect()
        {};
        bool post-condition(string pid)
        {return true}
};
```

Figure 50: Generated C++ Code of Class GateR2

```
GateR3()
{
requires curstate="toOpen";
}
```

Figure 51: Generated Larch/C++ interface specification of GateR3

```
class GateR3 : public transition
{
      private:

      public:
            GateR3()
            {
            name="GateR3";
            sstate="toOpen";
            dstate="opened";
            eventname="Up";
            };
            bool port-condition(string pid)
            { return true};
            bool enabling-condition(string pid)
            { return true};
            void effect()
            {};
            bool post-condition(string pid)
            {return true}
}
```

Figure 52: Generated C++ Code of Class GateR3

```
GateR4()
{
requires curstate="colsed";
}
```

Figure 53: Generated Larch/C++ interface specification of GateR4

```
class GateR4 : public transition
{
      private:

      public:
            GateR4()
            {
            name="GateR4";
            sstate="closed";
            dstate="toOpen";
            eventname="Raise";
            };
            bool port-condition(string pid)
            { return true};
            bool enabling-condition(string pid)
            { return true};
            void effect()
            {};
            bool post-condition(string pid)
            {return true}
};
```

Figure 54: Generated C++ Code of Class GateR4

Figure 55 shows the generated C++ code of subsystem *Trani-Controller-Gate* which with 5 Trains, 2 Controllers and 2 Gates. The properties *neme* and *Ports* of each reactive object are instantiated here. The communication links between compatible ports of interacting objects are also be given in it.

```
Class TrainControllerGate
{
      Gate Gate1, Gate2;
      Controller Controller1,Controller2;
      Train Train1, Train2, Train3, Train4, Train5;
      Gate1.set_name("Gate1");
      Gate2.set_name("Gate2");
      Gate1.ports =set⟨port⟩();
      Gate2.ports =set⟨port⟩();
      Controller1.set_name("Controller1");
      Controller2.set_name("Controller2");
      Controller1.ports = set⟨port⟩();
      Controller2.ports = set⟨port⟩();
      Train1.set_name("Train1");
      Train2.set_name("Train2");
      Train3.set_name("Train3");
      Train4.set_name("Train4");
      Train5.set_name("Train5");
      Train1.ports = set⟨port⟩();
      Train2.ports = set⟨port⟩();
      Train3.ports = set⟨port⟩();
      Train4.ports = set⟨port⟩();
      Train5.ports = set⟨port⟩();
      Gate1.ports.add(port("@S1","@S","Gate1","@G1","@G","Controller1"));
      Gate2.ports.add(port("@S2","@S","Gate2","@G2","@G","Controller2"));
      Controller1.ports.add(port("@G1","@G","Controller1","@S1","@S","Gate1"));
      Controller2.ports.add(port("@G2","@G","Controller2","@S2","@S","Gate2"));
      Controller1.ports.add(port("@P1","@G","Controller1","@C1","@C","Train1"));
      Controller1.ports.add(port("@P2","@G","Controller1","@C2","@C","Train2"));
      Controller1.ports.add(port("@P3","@G","Controller1","@C3","@C","Train3"));
      Controller2.ports.add(port("@P4","@G","Controller2","@C4","@C","Train3"));
      Controller2.ports.add(port("@P5","@G","Controller2","@C5","@C","Train4"));
      Controller2.ports.add(port("@P6","@G","Controller2","@C6","@C","Train5"));
      Train1.ports.add(port("@C1","@C","Train1","@P1","@G","Controller1"));
      Train2.ports.add(port("@C2","@C","Train2","@P2","@G","Controller1"));
      Train3.ports.add(port("@C3","@C","Train3","@P3","@G","Controller1"));
      Train3.ports.add(port("@C4","@C","Train3","@P4","@G","Controller2"));
      Train4.ports.add(port("@C5","@C","Train4","@P5","@G","Controller2"));
      Train5.ports.add(port("@C6","@C","Train5","@P6","@G","Controller2"))
}
```

Figure 55: Generated Code of the Subsystem: TrainControllerGate

Chapter 8

Conclusions

An important issue in real-time reactive system development is the implementation of the system that performs as intended in its environment. Due to inherent complexities in tracking and analyzing the requirements, arriving at a design that meets both the *functional requirements* and *timing requirement* is very difficult. In particular, when the size of the system is very large, the design and its validation become more difficult. TROMLAB environment provides TROM methodology for the design and a group of tools for design analysis. The implementation method discussed in this thesis adds an important component to TROMLAB, making the environment more complete with respect to life-cycle activities.

The implementation component has three sub-components:

- data type implementation;
- real-time execution model; and
- mapping from TROM design to Larch/C++ and C++ programs

Satisfying the *timing requirements* in the implementation is a challenging problem. In general, it depends on the hardware resources. We have assumed that the design stage has already taken the resource constraints. The implementation addresses concerns on the different functional and time sensitive capabilities of the physical and software components in the system, together with the time delay of message passing. The system is currently implemented in an uniprocessor environment. So, insufficient computing capacity may bring about some default, especially in the case when more than one event is delivered to the same object at the same time. Notice that, in a system two objects may be interacting while another object is executing an internal action. The sequential implementation uses interleaving semantics, which is technically correct; however, it may cause an adverse impact in a system run. When the size of the system becomes very large, the *partial order* of the events cannot be guaranteed in the implementation.

The transition specifications are mapped onto Larch/C++ interface specifications, and C++ programs are generated. The primary advantage is that the program can be tested with respect to the specification. This is achieved while the C++ program is run for a given set of environmental events. The result of a program fragment, a value or a status (state, attribute vector, reaction vector), should satisfy the postcondition of the Larch/C++ specification corresponding to the program fragment.

Future work include the following:

1. Integrate specification-based testing with the code generation;
2. Evaluate the performance of the generated code on different case studies;
3. Code reuse for class refinements and systems composed from other implemented (and tested) subsystems.

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, October 1996 (first draft), June 2000 (revised).

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

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

[AM98] V. S. Alagar and D. Muthiayen. Specification and verification of complex real-time reactive systems modeled in UML. Technical Report, Department of Computer Science, Concordia University, Montréal, June 1999.

[AOM00] V. S. Alagar, O. Ormandjeva and M. Zheng. Specification-Based Testing of Real-Time Reactive Systems. In *TOOLS USA* (to appear), Santa Barbara, CA, July 2000.

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

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

[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.

[KAR00] P. Karvelas. Schedulability analysis and automated implementation of real-time object-oriented design models. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, April 2000.

[LEA99] G. T.Leavens. Larch/C++ Reference Manual. http://www.cs.iastate.edu /~leavens/larchc++manual/lcpp_toc.html.

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

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

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

[Oana99] O. Popista. Rose-GRC translator: Mapping UML visual models onto formal specifications. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, March 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*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, New York, 1992. Springer Verlag.

[Pop99] F. Pompeo. A formal verification assistant for TROMLAB environment. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, March 1999. Under preparation.

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

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

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

[Sri99] V. Srinivasan. An intelligent graphical interface system for TROMLAB. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, December 1999.

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