INFORMATION TO USERS

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

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

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

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

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

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

IMPLEMENTING REAL-TIME REACTIVE SYSTEMS FROM OBJECT-ORIENTED DESIGN SPECIFICATIONS

Lizhong 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

August 2000

Abstract

Implementing Real-Time Reactive Systems from Object-Oriented Design Specifications

Lizhong Zhang

Real-time reactive systems are among the most difficult systems to design and implement because of their size and complex functional and timing requirements. TROMLAB is a framework for a rigorous development of consistent design that satisfy an authoritative specification of requirements, validate the design through simulation, and verify the design for safety properties through a formal verifier. This thesis adds one more significant component to TROMLAB by providing a methodology for automatic generation of code in real-time Java for reactive systems designed in TROMLAB framework. The correctness and efficiency of the implementation are illustrated on the implementation of the design for a generalized rail-road crossing problem, a bench-mark case study in the real-time systems community.

Acknowledgments

A warm thank you goes to my supervisor, Dr. V. S. Alagar, who guided this work with good advice, patience and understanding, and provided good technical support.

On a personal level, I thank my brother Liqiang, for his support and help. I wish to thank my parents for their financial and spirit support. Without their support, I cannot finish it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Real-time reactive systems

Reactive systems interact continuously with their environment through stimulus and response. Their stimulus-response behaviors are regulated by strict time constraints for the real-time aspect. Such systems are widely and increasingly used throughout society. Examples of such systems include air traffic control, nuclear power reactor, telecommunication system and strategic defense system. These systems must satisfy two important requirements:

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

Generally, the behavior of a reactive system is *infinite*: the process in a reactive system is usually continuously and non-terminatingly responding to the stimuli from its environment. The correct behavior of non-realtime systems is founded on the *functional correctness* of the result. By contrast, real-time reactive systems require both the functional correctness of the result and its *timing correctness*. Hence *safety* of such systems, which concerns prevention of risks to human life or property, relies on good analysis of the functional and timing properties of those systems. Nevertheless,

real-time reactive systems are large and complicated and consequently understanding or describing the behavior of such systems becomes very difficult.

Timed Reactive Object Model (TROM) formalism is introduced in [Ach95] for describing functional and timing properties of real-time reactive systems with formal notations. Formal notations are easy to understand and adequate to deal with complexity in modeling and designing. Further they can lead to rigorous validation and verification of the modeled systems. Moreover, tools for supporting TROM formalism have been developed. TROMLAB is a framework for applying TROM formalism and constructing real-time reactive systems in concordance with the process model. Figure 1 is an overall architectural view of TROMLAB.

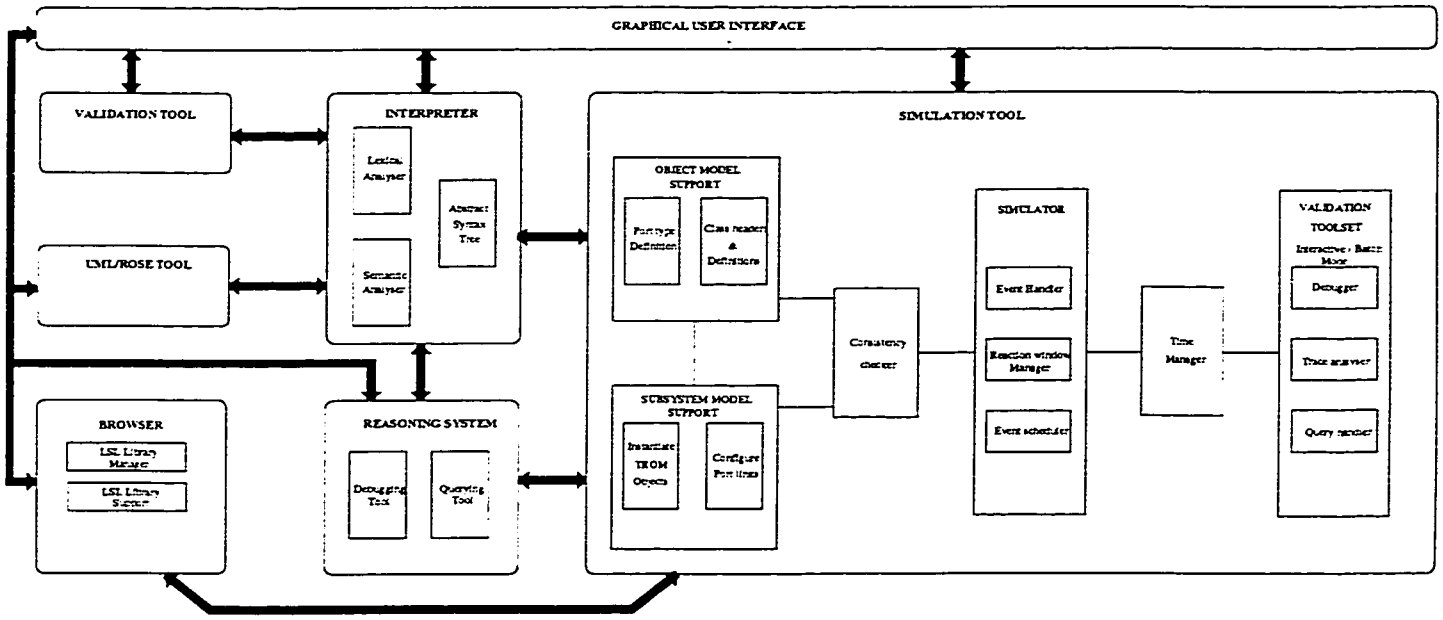


Figure 1: TROMLAB architecture

The following components of TROMLAB are currently operational:

- **Interpreter** - [Tao96] A parser, syntax checker and internal representation constructor;
- **Simulator** - [Mut96] A tool that simulate a subsystem based on the internal representation and enables a systematic validation of the specified system;
- **Browser** - [Nag99] A tool that help users navigate, query and access various system components for reuse during system development;

- **UML-RT Support** - [Oana99] A translator to generate TROM specification from Real-time UML;
- **Verification Assistant** - [Pop99] A tool to generate PVS theory from TROM specification for proving timing properties;
- **Graphical User Interface** - [Sri99] A visual modeling and interaction facility for a developer using the TROMLAB environment;
- **Reasoning System** - [Hai99] A tool to provide a means of debugging the system during animation by facilitating interactive queries of hypothetical nature on system behavior.

1.2 Research Goals

The objective of my research work is to give an object-oriented implementation methodology for implementing real-time reactive systems designed with TROM. Also a code-generation tool is developed in this thesis. TROM formalism is a kind of *Model-based Software Engineering* (MBSE) technique where those reusable resources are designed for flexibility and will not easily fail as changes occur in functionality, performance or technology. Accordingly, an object design model named *Time Reactive Object Model* is given in TROM formalism. As a result, an object-oriented implementation model which corresponds to the design model must be given in this thesis at the same time in order to generate application programs easily and make them reusable. Figure 2 shows my research goals in the context of TROMLAB, an environment designed and implemented at Concordia for a rigorous development of real-time development of real-time reactive systems.

The main contributions of this thesis are:

1. Providing an object-oriented TROM implementation model that matches with TROM design model.
2. Extending the abstract data model for making the code generation easy.
3. Developing TROM implementation support libraries.
4. Developing a code-generation tool.

Moreover, in this thesis Java is chosen for implementing abstract data libraries and TROM implementation support libraries. However, the standard Java language is not suitable enough for TROM environment since it lacks a standard thread scheduling algorithm. Thread synchronization becomes less tractable when the number of signals increases. Therefore, a *Real-Time Java*(RT-Java) class library for solving these problems is developed. The train-gate-controller case study, a benchmark example studied by real-time reactive system community, is implemented in RT-Java.

The structure of this thesis is as follows. Chapter 2 presents the TROM design methodology and the concepts that will be used in the later chapters. Chapter 3 presents an object-oriented implementation methodology for TROM. Chapter 4 describes the design details of the automatic code generation tool. Chapter 5 presents a case study using an example that demonstrates our implementation methodology and the usefulness of the tool. The thesis ends with Chapter 6 which presents the conclusions to be drawn from this thesis work and also presents the future work angle.

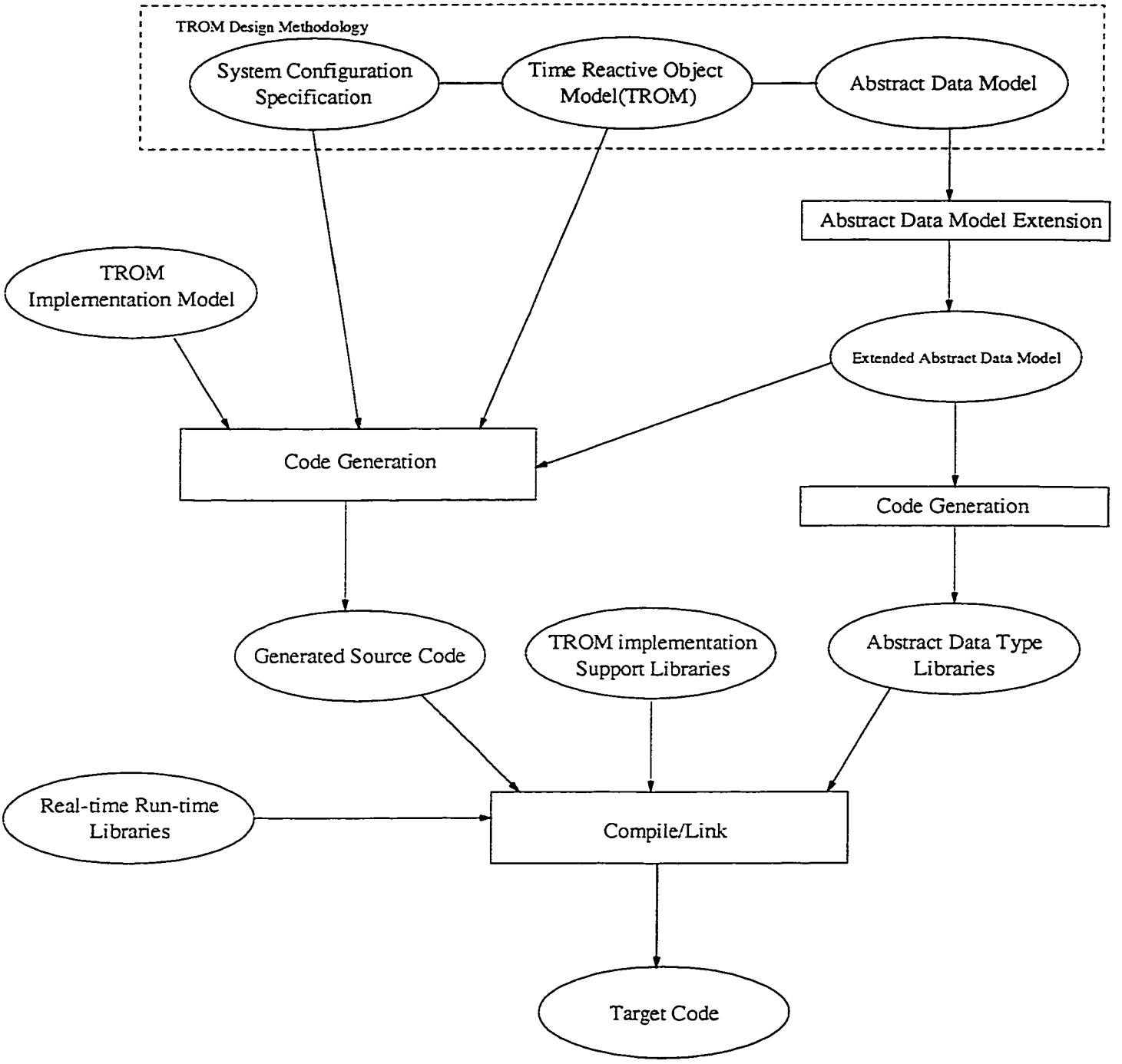


Figure 2: Research goals in the context of TROMLAB

Chapter 2

Reactive Model - TROM Methodology

2.1 Introduction

TROM methodology that is an object-oriented modeling technique for real-time reactive systems was introduced in [Ach95]. It introduces a three-tiered framework as the language for design specification. Figure 3 shows an overview of this methodology. The tiers independently specify systems configured with abstract data types, reactive classes, and reactive objects. This chapter introduces the basic concepts and terminologies used in the rest of this thesis.

2.2 First Tier - Larch Formalism

In the design framework of TROM, this tier specifies the data abstractions used in the class definitions of the second tier by means of one of the languages of Larch, the *Larch Shared Language* (LSL). An abstract data type is defined as LSL trait. Larch provides a two-tier approach to specification of program interfaces:

- In the *interface tier*, a Larch Interface Lauguage (LIL) is used to describe the behavior of a program module written in a specific programming language.
- In the *shared tier*, the Larch shared Language (LSL) is used to specify state-independent, mathematical abstractions that can be referred to in the interface

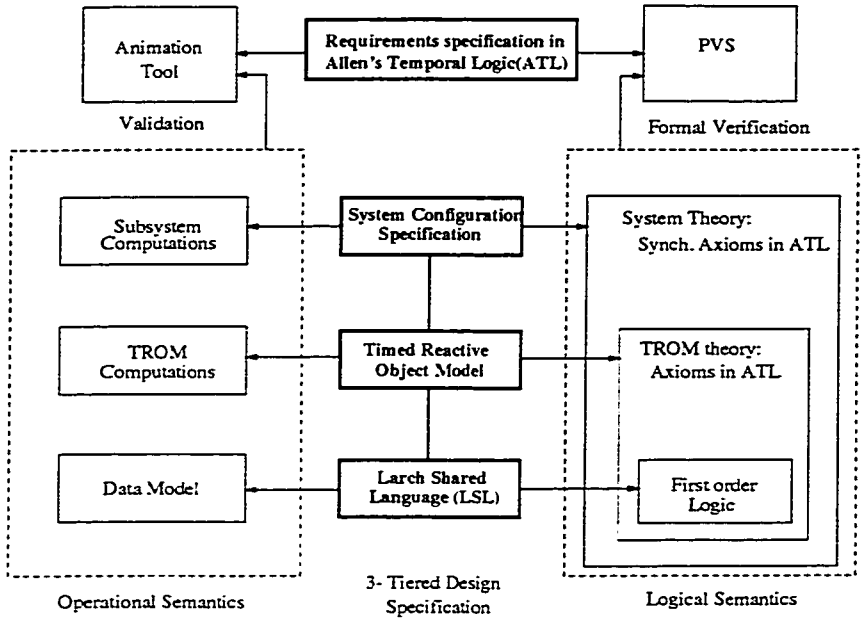


Figure 3: An overview of TROM methodology

tier.

In the present implementation of TROMLAB, only LSL traits are included. Figure 4 defines a trait that specifies *Set* data type.

```
Trait: Set(e, S)
    Include: Integer, Boolean
    Introduce:
            creat : ->S;
            insert : e, S->S;
            delete : e, S->S;
            size : S->Int;
            member : e, S->Bool;
            isEmpty : S->Bool;
            belongto : e, S->Bool;
end
```

Figure 4: Set trait

2.3 Second Tier - TROM Formalism

A TROM object has a single thread of control. Communication mechanism between TROMs is based on synchronous message passing, also known as *rendezvous.* A message passing between a TROM and its environment is represented by an *interaction.* An interaction of a TROM with its environment occur at a *port* associated with the TROM. Each port has a unique *port-type.* A *state* is an abstraction denoting an environmental information or a system information during a certain interval of time. A state can be either *simple* or *complex.* A complex state has an initial state and a set of simple and complex *substates.* A TROM class has a unique initial state. An *event* denotes an instantaneous activity. The events are classified into three types: *Incoming*, *Outgoing* and *Internal.* The attributes of a TROM class are of two kinds: (i) abstract data types imported from the first tier, and (ii) port types.

A formal definition of the different components of a reactive object as described above is presented next.

A reactive object is an 8-tuple $(\mathcal{P}, \mathcal{E}, \Theta, \mathcal{X}, \mathcal{L}, \Phi, \Lambda, \Upsilon)$ such that:

- $\mathcal{P}$ is a finite set of port-types with a finite set of ports associated with each port-type. A distinguished port-type is the null-type $P_\circ$ whose only port is the null port $\circ$.
- $\mathcal{E}$ is a finite set of events and includes the silent-event tick. The set $\mathcal{E} - \{\mathsf{tick}\}$ 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 $e \in (\mathcal{E}_{in} \cup \mathcal{E}_{out})$, is associated with a unique port-type $P \in \mathcal{P} - \{P_\circ\}$.
- Θ is a finite set of states. $\theta_0 \in \Theta$, is the *initial* state.
- $\mathcal{X}$ is a finite set of typed attributes. The attributes can be of one of the following two types: i) an abstract data type specification of 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,

- $\Phi_s : \Theta \rightarrow 2^{\Theta}$ associates with each state θ a set of states, possibly empty, called *substates*. A state θ is called *atomic*, if $\Phi_s(\theta) = \emptyset$. By definition, the initial state θ_0 is atomic. For each non-atomic state θ, there exists a unique atomic state $\theta^* \in \Phi_s(\theta)$, called the entry-state.
- $\Phi_{at} : \Theta \rightarrow 2^{\mathcal{X}}$ associates with each state θ a set of attributes, possibly empty, called the *active* attribute set. At each state θ, the set $\overline{\Phi_{at}}(\theta) = \mathcal{X} - \Phi_{at}(\theta)$ is called the *dormant* attribute set of θ.

- Λ is a finite set of *transition specifications* including λ_{init}. A transition specification $\lambda \in \Lambda - \{\lambda_{init}\}$, is a three-tuple : $< \langle \theta, \theta' \rangle; e(\varphi_{port}); \varphi_{en} \Longrightarrow \varphi_{post} >$; where:
  - $\theta, \theta' \in \Theta$ are the source and destination states of the transition;
  - event $e \in \mathcal{E}$ labels the transition; φ_{port} is an assertion on the attributes in $\mathcal{X}$ and a reserved variable pid, which signifies the identifier of the port at which an interaction associated with the transition can occur. If $e \in \mathcal{E}_{int} \cup \{\mathsf{tick}\}$, then the assertion φ_{port} is absent and e is assumed to occur at the null-port o.
  - φ_{en} is the enabling condition and φ_{post} is the postcondition 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 and it implicitly specifies the data computation associated with the transition.

  For each $\theta \in \Theta$, the silent-transition $\lambda_{s\theta} \in \Lambda$ is such that,

  $$\lambda_{s\theta} : \langle \theta, \theta \rangle; \mathsf{tick}; true \Longrightarrow \forall x \in \Phi_{at}(\theta) : x = x';$$

  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 $\upsilon_i \in \Upsilon$ 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.

The grammar of the formal class specification, based on the above formalism is given in Appendix A. Figure 5 shows the template for a class specification.

```
Class < name >
    Events:
    States:
    Attributes:
    Traits:
    Attribute-Function:
    Transition-Specifications:
    Time-Constraints:
end
```

Figure 5: Template for System Configuration Specification.

Figure 6 shows a TROM class description for an arbiter specification. An arbiter allocates shared resources to processes requesting them. The arbiter modeled in Figure 6 puts the requests for a resource received from processes into a set and allocates the resource to the next process waiting in the set. The specification uses the functions *insert* and *delete* from the signature of the trait *Set(@U, USet)*, imported into the *Arbiter* class, to add and delete requests mode 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. Figure 7 shows a TROM class for a user of the services of the arbiter. The attibute *arb* in class *User* denotes the most recent port at which a resource was received.

2.4 Composite Classes

A TROM object has a single thread of control. This simplifies system designs, but limits the scope of containing design complexity. *Composite class* is introduced to

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:
  $allot \mapsto \{rqSet\}$; $wait \mapsto \{rqSet, hold\}$;
 Transition-Specifications:
  R_1 : $\langle idle, allot\rangle$; *Req?(true)*;
   $true \Longrightarrow rqSet' = insert(\text{pid}, \{\})$;
  R_2 : $\langle allot, wait\rangle$; *Grant!*(pid $\in$ *rqSet*);
   $true \Longrightarrow rqSet' = delete(\text{pid}, rqSet)) \wedge (hold' = \text{pid})$;
  R_3 : $\langle allot, allot\rangle$, $\langle wait, wait\rangle$; *Req?(not* pid $\in$ *rqSet)*;
   $true \Longrightarrow rqSet' = insert(\text{pid}, rqSet)$;
  R_4 : $\langle wait, allot\rangle$; *Ret?* (pid = *hold*);
   $\neg$ *isEmpty(rqSet)* $\Longrightarrow rqSet' = rqSet$;
  R_5 : $\langle wait, idle\rangle$; *Ret?* (pid = *hold*);
   isEmpty(rqSet) $\Longrightarrow$ *true*;
 Time-Constraints:
  TC_1 : (R_1, *Grant*, [0,2], $\emptyset$)
  TC_2 : (R_4, *Grant*, [0,2], $\emptyset$)
end

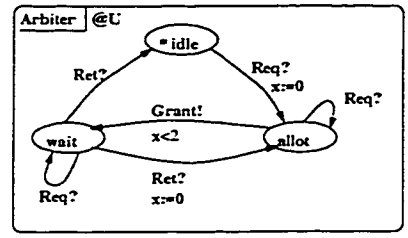


Figure 6: Class definition of Arbiter

Class *User* [@A]
 Events: *Req!A, Grant?A, Ret!A*
 States: **idle, wait, hold*
 Attributes: *arb:@A*
 Attribute-function:
  $wait \mapsto \{arb\}$;
 Transition-Specifications:
  R_1 : $\langle idle, wait\rangle$; *Req!(true)*;
   $true \Longrightarrow arb' = \text{pid}$;
  R_2 : $\langle wait, hold\rangle$; *Grant?*(pid = *arb*);
   $true \Longrightarrow true$;
  R_3 : $\langle hold, idle\rangle$; *Req!(* pid = *arb)*;
   $true \Longrightarrow true$;
 Time-Constraints:
  TC_1 : (R_2, *Ret*, [0,20], $\emptyset$)
end

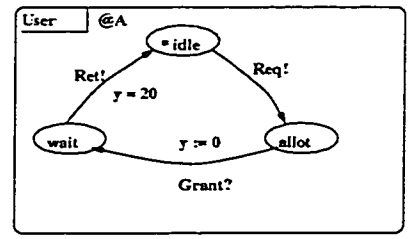


Figure 7: Class definition of User

minimize design complexity and to promote modularity at subsystem level. TROM classes and composite classes can be composed to obtain a new composite class by *glueing* the compatible port types. Figure 8 show the syntax for describing a composite classes.

```
CompositeClass<identifier>[<listofport-type>]
    Incarnations:
    Connectors:
end
```

Figure 8: Template for Composite Class Construction

The template includes the keyword CompositeClass introducing the name of the composite class, and sections labeled with the keywords Incarnations, and Connectors. An incarnation of a class is the class specification in which the port-type parameters may be renamed. The Incarnations section lists the incarnations of classes that participate in a composition. When several incarnations of a class participate in a composition, the port types that become available for external communication must sometimes be *renamed* in order to resolve ambiguity in port-type identifiers. Conflict in port type identifiers may also arise when incarnations of different composite classes participate in a composition. The Connectors section lists the connectors that glue compatible ports for internal communication between the components. Figure 10 gives the specification of the composite class shown in Figure 9.

2.5 Third Tier - SCS

A *System Configuration Specification* (SCS) describes the system architecture by succinctly specifying the interaction relationship that can exist between the objects in a system. The template in Figure 11 shows the syntax for subsystem configuration specifications.

The syntax includes the keyword sf SCS to introduce the identifier for the system, and sections labeled with the keywords Include, Instantiate, and Configure. The Include clause is for importing other subsystems. The Instantiate clause defines reactive objects by parametric substitutions to cardinality of ports for each port type, and

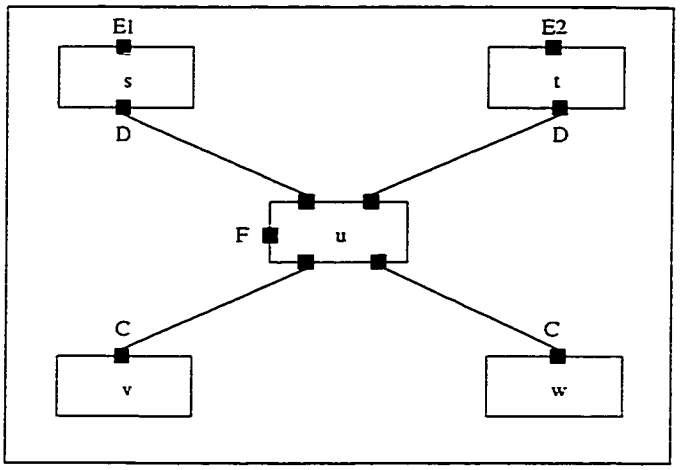


Figure 9: A composite class with five classes

initializing the values of attributes in the initial state of the object. The Configure clause defines a configuration obtained by composing objects specified in the Instantiate clause and the subsystem specifications imported through the Include clause. Figure 12 shows a subsystem configuration for an arbiter system involving two users and one arbiter. The *Arbiter* objects have two ports of type $@U$; each *User* object has one port of type $@A$.

2.6 Design Refinement

A design can be refined by adding more details, which may require adding more states, transitions, and strengthening time constraints. The refined design, which is more detailed than the original design, must preserve essential properties of the original design and aid the implementation process. However, the design obtained from an unconstrained inheritance principle does not guarantee the preservation of properties in the derived TROM. Towards remedying this, three forms of constrained inheritances based on *subtyping* are introduced in TROM.

2.6.1 Behavioral Inheritance

The TROM object $\mathcal{A}$ obtained by refining the TROM object $\mathcal{A}'$ according to the following criteria inherits the behavior of TROM $\mathcal{A}'$.

CompositeClass *Class5*[@E_1, @E_2, @F]
    Incarnations:
        s: *Class1*[@E_1 *for* @E, @D]
        t: *Class1*[@E_2 *for* @E, @D]
        u: *Class2*[@A, @B, @F]
        v, w: *Class3*[@C]
    Connectors:
        $s.@D \leftrightarrow u.@A$
        $t.@D \leftrightarrow u.@A$
        $v.@C \leftrightarrow u.@B$
        $w.@C \leftrightarrow u.@B$
end

Figure 10: Specification for Composite Class *Class5*

Subsystem < *name* >
    Include:
    Instantiate:
    Configure:
end

Figure 11: Template for System Configuration Specification.

Subsystem *ArbiterSystem*
    Include:
    Instantiate:
        ar_1 :: *Arbiter*[@U : 2].*Create*();
        us_1, us_2 :: *User*[@A : 1].*Create*();
    Configure:
        $ar_1.@u_1 \leftrightarrow us_1.@a_1$;
        $ar_1.@u_2 \leftrightarrow us_2.@a_2$;
end

Figure 12: Arbiter System

1. *Attribute redefinition:* the data model of an attribute may be redefined, provided there exist coercion functions for each attribute from the refined trait to the original trait.

2. *Transition redefinition:* Redefinition of an inherited transition specification may be done such that:

   - The post-condition may be strengthened.
   - The port-condition of a transition involving any event $e \in \mathcal{E}_{out}$ may be strengthened.
   - The enabling-condition of a transition involving any event $e \in \mathcal{E}_{out}$ may be strengthened.
   - The initial attribute-constraint, φ_{init} may be strengthened.

3. *Time-constraint redefinition:* the minimal time delay may be increased or the maximal time delay may be decreased.

2.6.2 Extensional Inheritance

The goal of extensional inheritance is to provide design refinements that preserve behavior. The TROM object $\mathcal{A}$ obtained by refining a TROM object $\mathcal{A}'$ satisfying the following constraints is an extensional inheritance of $\mathcal{A}'$.

1. Possible Redefinitions:

   - Any redefinition as permitted in Behavioral inheritance.
   - The source state θ of a transition may be redefined to any substate of θ.
   - The enabling-condition of any inherited transition may be strengthened.
   - The port-condition of any inherited transition may be strengthened.

2. Possible Additions:

   - *Event addition:* New events may be added, enriching inherited port-types.
   - *State addition:* A new state may be added only to make an existing simple state as a complex state.

- *Attribute addition:* New attributes and traits may be added.
- *Transition addition:* An added transition can have only new events and new states. The superstate of the source state and the destination state should be the same.
- *Time-constraint addition:* An added time-constraint can only constrain a new event.

2.6.3 Polymorphic Inheritance

The TROM $\mathcal{A}$ is a polymorphic inheritance of the TROM object $\mathcal{A}'$, if the following conditions are satisfied:

1. Possible Redefinitions:
   - Any redefinition as permitted in Behavioral inheritance.
2. Possible Additions:
   - *Port addition:* New port-types may be added. This necessarily introduces new events.
   - *State addition:* New states may be added without decomposing any atomic state.
   - *Attribute addition:* New attributes and traits may be added.
   - *Transition addition:*
     - An added transition can only involve new events.
     - An added transition involving an event $e \in \mathcal{E}_{in}$ can only have newly added states for both source and destination states.
   - *Time-constraint addition:* An added time-constraint can only constrain an new event and can only be triggered by a new transition.

See Achuthan [Ach95] for examples on design refinements. A detailed design obtained through one or more kinds of refinements is more suitable for code generation.

Chapter 3

Implementing TROM Models

3.1 Introduction

An object-oriented implementation of reactive systems specified according to the TROM notations is discussed in this Chapter. The goal of the implementation phase is to automate the code generation process so that the resulting program is correct and efficient. The program is correct in the sense that the functional and timing constraints in the implemented program is consistent with the implicit functionality specified in the transition specifications and the timing constraints specified for the time-constrained events. The efficiency comes in two forms - the code is junk-free and the execution of the program conforms to the timing requirements in the presence of appropriate resources. A good implementation must also seamlessly mesh with the design. This quality depends upon the expressive power of the programming language that can implement the design notations.

3.2 Environment Assumptions

Implementing TROM models require transforming TROM model specifications into implementations based on concepts supported by real-time implementation environments. Unfortunately, there does not exist a general real-time implementation environment for all of real-time reactive systems. In this section, we will give the basic

assumptions about the target environment and the implementation programming language, which suits our implementation methodology.

An implementation environment consists of two parts: a programming environment (including a programming language) and a platform run-time environment (usually an operating system kernel on a specific hardware platform). The primary assumption about this environment is that it supports some type of *concurrent programming paradigm*. This paradigm may be supported directly in the language (as in Smalltalk) or it may be supported through a suitable run-time library. For example, the C++ language does not support concurrency, but it is still possible to write concurrent application in C++, provided that a multitasking run-time library is included with the application. The following assumptions should be satisfied by this environment as well:

- Programming Language — An object-oriented language will be used for implementation.
- Multitasking — Some form of lightweight concurrency unit or thread is supported by the underlying kernel.
- Thread Synchronization — In TROM model, thread synchronization is unnecessary, since there is no shared memory between TROM objects.
- Interprocess Communication — An asynchronous message passing is supported at least.
- Scheduling — Priority scheduling is supported.
- Memory Model — All memory references are local to a TROM object.

3.3 Real-Time Run-Time Library

There are many programming languages that are used all over the world. Many major programming languages, such C++, do not provide any facilities for concurrent programming. Some extended languages, such as Smalltalk, and Java, provide high-level concurrent programming facilities. Communication mechanisms in those languages that support concurrent programming may be different since there is no

general standard on the "right" facilities for concurrent programming. For example, communication mechanisms should be based on message passing, updating shared data or both paradigms? Moreover, a program written in a specific programming language may have different behaviors in different operating systems since thread scheduling may depend on the operating system. So the implementation relating to thread and interprocess communication depends on the programming language chosen for implementing the system, the operating system selected for running the system, and the hardware environment.

As mentioned above, the purpose of this thesis is not to give a general standard on the facilities for concurrent programming. What we want is to separate high-level implementation from the environment. We develop a real-time run-time library over the programming language chosen for implementing the system. In this library, an object interface for creating a thread and starting a thread should be given and an interprocess communication mechanism should be provided too. Moreover, it would be done to make the system run exactly the same manner in different operating systems as in this library if it is possible. This library can be re-used in the implementations of all of real-time reactive systems that are implemented with this programming language.

3.3.1 Real-time Reactive Model

A reactive model at the implementation level, that makes reactive systems have the same behaviors in different operating systems is shown in Figure 13. In this model, a *super thread* has the highest task priority, and takes control of task scheduling and message passing of other threads. Every thread controlled by the super thread is a TROM reactive object. When a thread wants to send an event to another thread, this event is sent to the *outgoing message queue.* The super thread will send this event to the *inner event queues* of both the sender and the receiver, and decide the executed sequence of threads. If the algorithm of task scheduling and message passing is given, the behaviors of reactive systems will become independent of any programming language and operating system. A *global clock* is defined for providing a logical clock to every thread in this model as well.

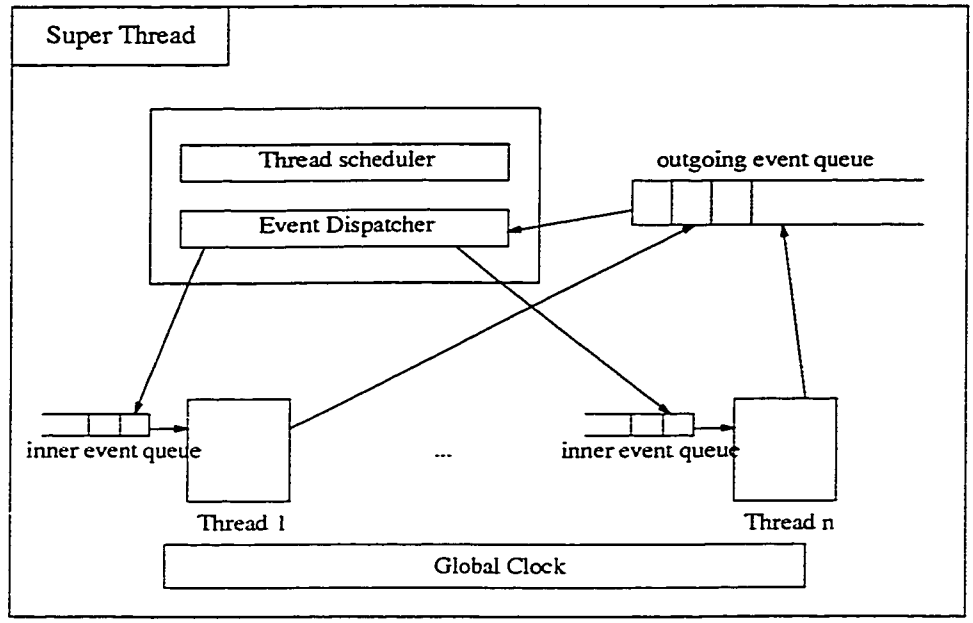


Figure 13: A Reactive Model

3.3.2 Task Scheduling and Message Passing Algorithm

Task scheduling is a major feature of real-time reactive systems. There are two ways to classify schedulers. A scheduler can be categorized as *static* or *dynamic*. Also it can be categorized as *non-preemptive* or *preemptive*. Static schedulers create the task execution pattern, or *schedule*, off-line and the latter is then used to dispatch tasks at run-time. In contrast, dynamic schedulers determine the schedule online, based on specific task characteristics. In a non-preemptive scheduler, a scheduled task keeps the processor until it decides to relinquish it voluntarily. On the other hand, in a preemptive scheduler, a scheduled task may be suspended, or *preempted*, so that the processor can be allocated to another task.

There are some task scheduling algorithms that may be appropriate for our reactive model, but they are not the best choice for the TROM model. In TROM model, communication mechanism between TROMs is based on synchronous message passing, also known as *rendezvous*. Therefore, the sender of a message should react with this message after the receiver of this message finishes its reaction of this message, or both of the sender and the receiver react with this message together. Moreover, messages happened in a TROM reactive system are *partially ordered*. This implies that it is unnecessary that two messages should have an executed order. The scheduling

algorithm for TROM should satisfy the requirement of synchronization and keep the partial order of messages. A non-preemptive dynamic scheduling algorithm combined with message passing is introduced later in this section.

There exists a two-level task priority in our scheduling algorithm, which is named TROM *scheduling algorithm.* The threads that have a lower priority imply these reactive objects are in a stable state that the reactive objects are waiting for the event from the environment or will fire an event when some conditions are satisfied. The threads that have a higher priority mean these reactive objects have an event to deal with. The primary assumption of this algorithm is that the computation time of a state or a transition, which is triggered by an event, must be less than the time slice assigned by the super thread. In another word, every thread only finishes a computation of a state or a transition in a time slice. Therefore, under a uni-processor environment, the physical time of a logical clock time unit must be bigger than (threads' number * time slice + communication time + scheduling time).

Initially, all of the reactive objects in a reactive system just have lower priority. They are put into the lower priority task queue. When one of them fires an event, the super thread will pick up this event. First, it will find the receiver of the event in the lower priority task queue. Then it will put this event into the inner message queue of the receiver and remove the receiver from the lower priority task queue and append it into the higher priority task queue. Later it will find the sender of the event in the lower priority task queue. Finally it will put this event into the inner message queue of the sender and remove the sender from the lower priority task queue and append it into the higher priority task queue. When a thread finishes its execution, the super thread will suspend this thread, append it into the lower priority task queue, and try to get the next task from the higher priority task queue. If the higher priority task queue is empty, the super thread will get the next task from the lower priority task queue. Then remove it from the queue and resume this thread. Therefore, we can conclude the receiver and the sender are synchronous. The above description is just the simplest case in the system. The real algorithm is more complicated than it is presented. The pseudo code of the whole algorithm is shown in Figure 14. This algorithm suits the uni-processor environment.

```
while true
    /* Message Passing */
    for i = 1 to Messages' Number in the outgoing message queue step 1
        receiver = receiver of Message[i];
        sender = sender of Message[i];
        j = 1;
        while j < i
            if receiver = receiver of Message[j]
                or receiver = sender of Message[j]
                or sender = receiver of Message[j] then
                break;
            endif
            j = j + 1;
        end while

        if j == i then
            sender-task = 0;
            receiver-task = 0;
            for k = 1 to tasks' number in the lower priority task queue step 1
                if task's name = receiver then
                    receiver-task = k;
                endif

                if task's name = sender then
                    sender-task = k;
                endif

                if sender-task <> 0 and receiver-task <> 0
                    put message into the inner message queue of task[receiver-task];
                    remove this task from the lower priority task queue;
                    append this task into the higher priority task queue;

                    put message into the inner message queue of task[sender-task];
                    remove this task from the lower priority task queue;
                    append this task into the higher priority task queue;

                    remove the message from the outgoing message queue.
                    break;
                endif
            endfor
        end if
    end for

    /* Task Scheduling */
    if the higher priority task queue is not empty then
        currentTask = the first task in the higher priority task queue;
        remove this task from queue;
    else
        currentTask = the first task in the lower priority task queue;
        remove this task from queue;
    endif
    resume currentTask;
    sleep;

    /* Activated by currentTask */
    append currentTask into the lower priority task queue;
end while
```

Figure 14: Task Scheduling and Message Passing Algorithm

3.3.3 A Real-time Run-time Library of Java

Java is chosen as the programming language for the case study since it is a pure object-oriented language. A real-time run-time library of Java is developed in the thesis. The library has 4 major components:

RTSuper - implements a super thread which takes control of task scheduling and message passing services.

RTThread - implements a general thread.

TimeManager - implements a global clock for providing the logical time.

MessageQueue - implements a message queue.

If some other object-oriented programming languages are used, the detail of implementation may be different. All of the methods we provide can be applied with minor changes.

3.4 Mapping TROM Models into RT-Environment

After the real-time run-time library is provided, the rest of our work for implementing TROM models is to map TROM models into this real-time environment (RT-Environment). In TROM methodology, there is a three-tiered design framework as the language for design specification. This three-tiered design framework describes the system level architectural details and the component level behavioral details by means of formal notations. Therefore, to translate the design seamlessly, our methodology also has a three-tiered implementation framework. Figure 15 shows the relationship between design framework and implementation framework.

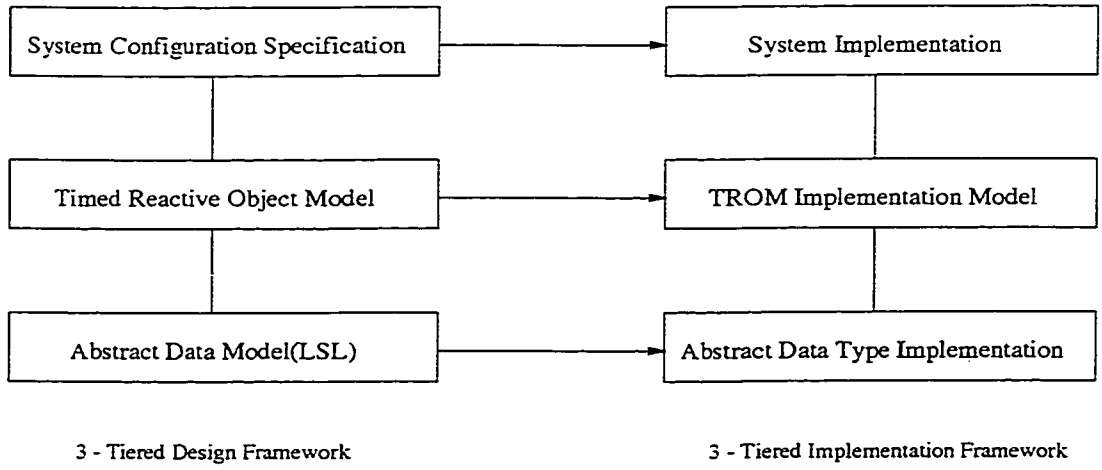


Figure 15: Relationship between design framework and implementation framework

In the three-tiered implementation framework, the system that consists of reactive objects, the reactive object classes, and abstract data types used in objects are implemented in separate tiers. The lower tier implements *Abstract Data Model*. The

middle tier implements *Timed Reactive Object Model.* The top-most tier implements *System Configuration Specification.* For implementing a system more efficiently and quickly, we have to give some rules that make the implementations of the tiers to have low-coupling among themselves.

3.4.1 Abstract Data Type Implementation

This tier implements *Abstract Data Model.* Abstract Data Model specifies the data abstractions used in the system by means of one of the languages of Larch, the *Larch Shared Language* (LSL). An abstract data type can be defined as LSL trait. If we directly implement an abstract data type from a LSL trait, we will get various implementations from different programmers. The programmers who implement the second tier cannot begin their work until the implementation of this tier is finished. So we provide interface specifications and function specifications written in the programming language of the implementation before the implementation begins.

Fortunately, Larch has an *interface tier*, where a Larch Interface Language (LIL) is used to describe the interface and behavior of a program module written in a specific programming language. An interface specification, which defines an interface between program components, is written from the point of view of clients who will use the module. The specification of a function in the interface documents its behavior. This can be understood without reference to other functions in the interface. The body of a function consists of a number of clauses. Most function specifications contain *requires*, *modifies*, and *ensures* clauses. The *requires* clause defines *pre-conditions* on the state and parameters. The *ensures* clause defines *post-conditions* on the state and parameters. The *modifies* clause states what a function is allowed to change. If there is no modifies clause, then nothing may be changed in this function.

Since LIL is formal, we suggest to extend the abstract data model with Larch Interface Language. When the language for the implementation is decided, we suggest to select the specific Larch Interface Language for this language to describe the interface and behavior of a program module. Therefore, the programmers of this tier can focus on how to implement the behavior and the programmers of the second tier will have the interface and behavior descriptions of abstract data types before abstract data models are implemented. Also, the reusability of the implementation of abstract data

types is increased. Furthermore, because Larch Interface Language is using a *pre-condition* and *post-condition* method to describe the behavior of a program module, test cases can be generated following the behavior description and the verification of this program module may be possible. In TROM design model, transition specification is described by using the assertions on the attributes, which are abstract data types and port types. These assertions are either between attributes or using function signatures among abstract data types in Larch. If every signature has an independent function which implements it, automatic code generation for transition may be possible.

3.4.2 TROM Implementation Model

This tier implements *Timed Reactive Object Model.* In TROM design model, a reactive object is modeled as a TROM. A TROM is a finite state machine augmented with ports, attributes, logical assertions on the attributes, and timing constraints. Therefore, an implementation of a reactive object is an implementation of a finite state machine. This implementation must satisfy the augments with ports, attributes, logical assertions on the attributes, and timing constraints as well.

In TROM design model, a TROM consists of 8 elements, such as a set of ports, a set of events, a set of states, a set of attributes, a set of traits, a set of attribute functions, a set of transitions, a set of time constraints. These 8 elements make the function and timing properties of a TROM more precisely. But not all of these 8 elements are the attributes of a TROM reactive object in the implementation view, since some of these 8 elements describe the implied properties of a TROM reactive object, such as a set of attribute functions, a set of events and a set of traits. Therefore, a TROM reactive object in our implementation model consists of these elements:

- a set of ports
- a set of states
- a set of typed attributes
- a set of transitions
- a set of time constraints

High Level Behavior

In general, the basic elements of a finite state machine are events, states and transitions. An event can trigger a transition. A transition is an action, which is sending messages, changing encapsulated data values, and so on. An action also can be taken in a state. A state machine is classified as *Mealy machines* if actions are associated with transitions and as *Moore machines* if actions are associated with states. TROM model have characteristics of both Mealy machines and Moore machines.

Actually, finite state machines in TROM are extended by adding ports and timing constraints. Two finite state machines interact with each other through their ports. Timing constraints regulate the occurrence of events. But the behavior mode of a TROM finite state machine is still the same as a general finite state machine. In other words, actions are still taken in transitions and states.

In our scheduling algorithm, the computation time of transitions and states is restricted by the size of a time slice. If the computation time of a transition is longer than a time slice, this transition can divide into several transitions that everyone fits in a time slice. If the computation time of a state is longer than a time slice, this state can be refined as a complex state. For example, figure 16 shows a state diagram of a printer. A printer is idle when it is waiting for a job. There should be some code for checking whether a new job is coming. Assume the execution time of this code is bigger than a time slice. The idle state can be refined as a complex state. The execution time of every substate of the new idle state is less than a time slice. The new state diagram is shown in figure 17. Actually, the printing state can be refined too.

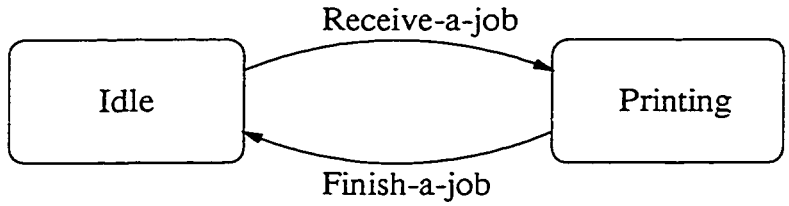


Figure 16: State Diagram of A Printer

Furthermore, the types of actions of a transition and actions of a state are different in TROM design model. Actions of a transition may change some values of attributes

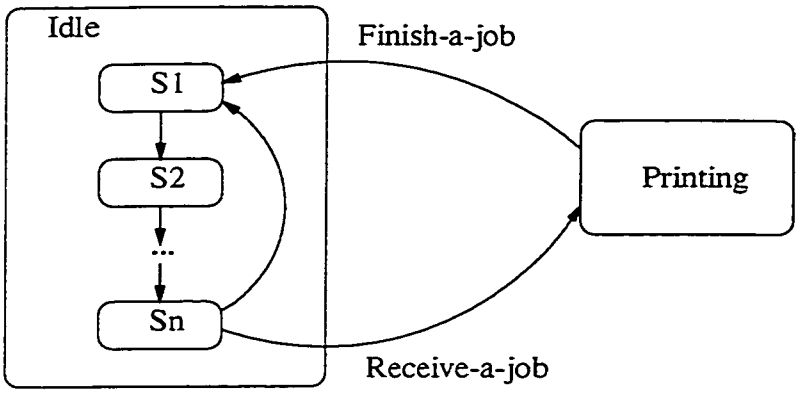


Figure 17: Refinement of State Diagram of A Printer

in a reactive object and the current state of the reactive object. Actions of a state may not change any values of attributes in a reactive object and just wait for the environment event or fire a event when some conditions are satisfied. Therefore, states are considered as *choicepoints*. That is, the next state can only be determined after some preliminary calculations are performed or some environments are changed.

Combining with the above issues, Figure 18 graphically shows the implementation behavior of a TROM reactive object. In a time slice, a TROM reactive object will check whether an event has been received. If so, this event will trigger a transition, and actions of this transition are taken. If not, actions of current state are taken.

Although events are sent to the outgoing queue and are received from the inner queue in the reactive model, events are still sent/received through ports in our implementation model. Therefore, "check event" means checking ports.

When an event arrives, a search of candidate transitions takes place to determine which one will be selected. Transitions are evaluated sequentially. A transition that satisfies all of the conditions shown as below will be selected:

- The source state of this transition is the current state of the reactive object.
- The triggering event of this transition is the same as the arrived event.
- The port at which the event arrived must satisfy the port condition of this transition.
- The values of attributes in the source state of the reactive object must satisfy

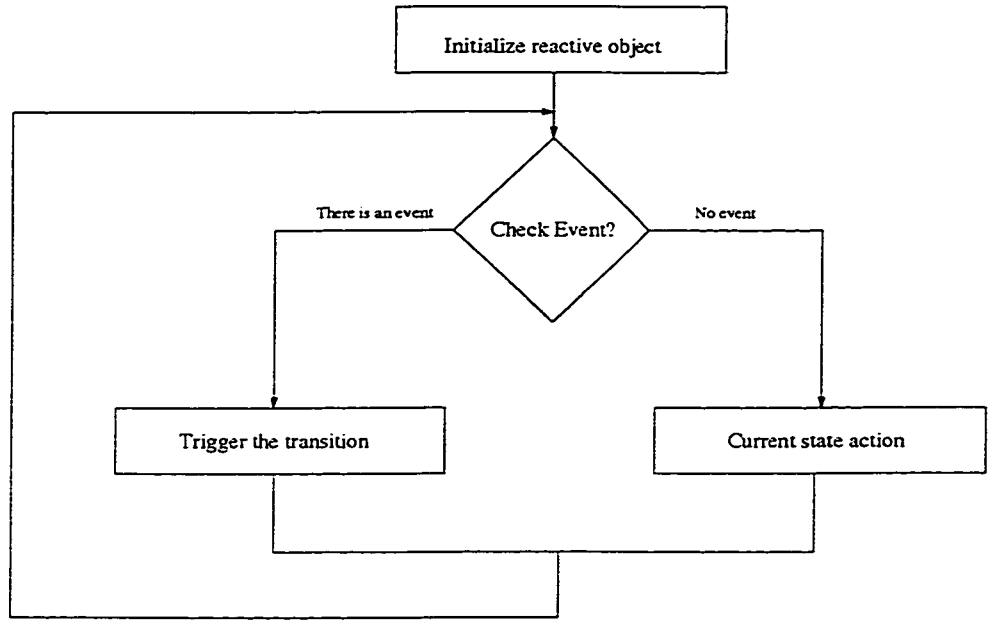


Figure 18: Behavior of a TROM reactive object

the enable condition of this transition.

The search terminates and the actions of this transition are taken. If no transitions can be satisfied by the event, the event is discarded and the state of the reactive object remains unchanged.

In implementation view, a timing constraint is a *timer* which provides the occurrence of a reactive event. This event should occur in a certain state of a TROM. Therefore, it may be necessary to check whether a timer is satisfied in actions of a state. Unconstraint internal events also occur in state actions.

Common Reactive Class

As discussed above, every TROM reactive object has the same high level behavior. Only the detailed behaviors, actions of transitions and states, are different. Also every TROM reactive object has some common attributes. A *Common Reaction Class* (CRC), which has the common attributes of reactive objects and implements the same high level behavior of reactive objects, is introduced in this section.

In TROM design model, a TROM reactive object has a single thread of control. Therefore, a CRC is a subclass of real-time thread (RTThread) which is defined in the

real-time run-time library. Figure 19 shows the class diagram of CRC. Some abstract data types are introduced as shown below:

- Transition - the abstract data type for transition.
- Transitions - a subclass of set, defines a set class for transition.
- State - the abstract date type for state.
- TimeConstraint - the abstract date type for time constraint.
- TimeConstraints - a subclass of set, defines a set class for time constraints.
- Port - the abstract data type for port.
- Ports - a subclass of set, defines a set class for ports.
- Event - the abstract date type for event.

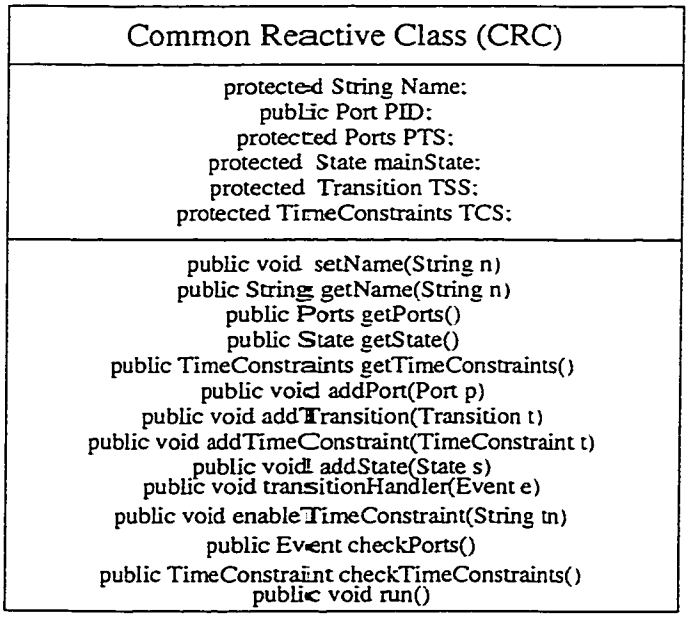


Figure 19: Class diagram of Common Reactive Class

Then attributes of a reactive object in our implementation model, such as a set of transitions, a set of states, a set of ports, and a set of timing constraints, are defined as date members of CRC. CRC is the kernel of our implementation model. Every

TROM reactive object class is a subclass of CRC and has different values for the data members. Every TROM reactive object class can also have its own data members.

The data member *Name* is for keeping the reactive object's name. The date member *PID* keeps the most recently used port. These two are the common attributes of TROM reactive objects.

The descriptions of methods in CRC are shown below:

- *setName, getName, getPorts, getState, getTimeConstraints, getTransitions:* provide the access points for those date members.
- *addState, addTransition, addPort, addTimeConstraint:* provide methods respectively for increasing state, transition, port, and time constraint.
- *checkPorts:* implements checking event.
- *transitionHandler:* implements selecting a transition and take the actions of the transition.
- *enableTimeConstraint:* implements starting timer.
- *checkTimeConstraints:* check whether a timer is satisfied.
- *run:* implements the high level behavior of reactive objects.

Event

Although event is not considered as an element of a reactive object in our implementation model, the abstract data type *event* still needs to be provided because a TROM reactive object is event-driven. An event has its name. For a parameterized event, a set of parameters should be the date member of event. Figure 20 shows the class diagram of event.

Those functions, such as *setName*, *getName*, *setPTS*, and *getPTS*, provide the access points for the data members of event.

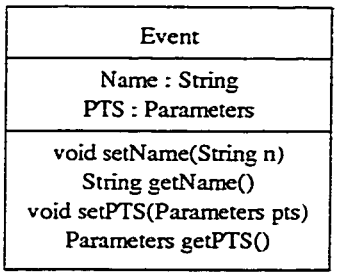


Figure 20: Class diagram of event

Attribute

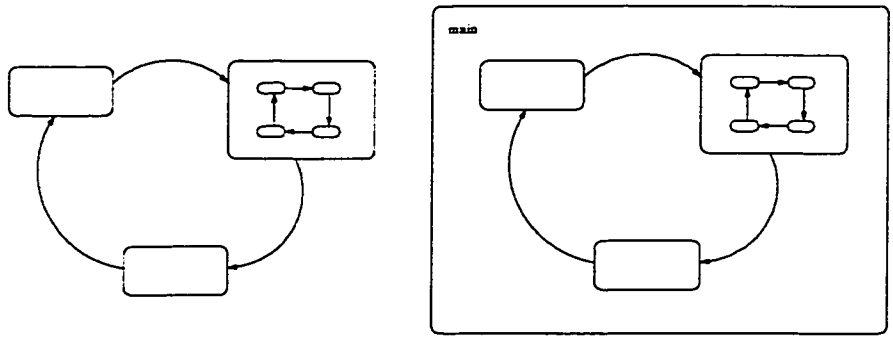
Actually, we can define a set of attributes as a data member in CRC in the same way as we define other elements. But the drawback of this approach is that it will decrease the performance of reactive objects since there is no way to access an element in a set by the name of the element directly. Instead, we define those attributes as public data members of the reactive object class. Therefore, these attributes can be accessed directly in order to get better system overall performance.

State

In our implementation model, the hierarchy of nested states is supported. So there are two types of states: *atomic* and *complex*. To simplify the implementation, the whole reactive object can be considered as a big complex state, which is named "main". For example, in Figure 21, A is the original design and B is the state diagram which is used in the implementation. Therefore, the current state of a reactive object becomes the current state of "main". Moreover, if the current state of "main" is complex, this state also has its own current state. So the nested states are implemented. Figure 22 shows the class diagram of State.

The descriptions of the data members and the methods of port class are shown as below:

- *Name:* the name of state.
- *Status:* a Boolean value to check whether this state is a initial state or not.



(A) (B)

Figure 21: Reconsider the state diagram

- *currentState:* the current substate of a state. If this state is atomic, it will be equal to null.
- *Substates:* a set of states which are the substates of this state.
- *setName, getName, setStatus, getStatus, setcState, getcState, setSubstates, getSubstates:* access points for the data members.
- *Atomic:* returning true when this state is atomic.
- *initialcState:* setting the initial substate of this state as the currecnt.
- *addState:* adding a new substate.
- *replaceState:* replacing the old substate with the new substate.
- *run:* a virtual function for actions of the state.

We do not allow any action to take place in the complex state, since it is impossible to decide whether it is the action of this state or the current substate of this state. Furthermore, if there are any actions in the complex state, we can always refine this state, make the actions of this state to be the actions of one of the substates of this state. Every state of a reactive object is a subclass of State class, and provides the implementation of the actions of the state.

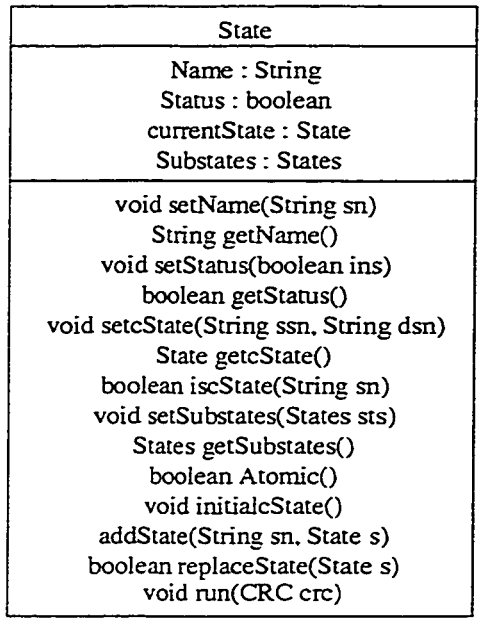


Figure 22: Class diagram of State

Port

A port is the only access point for a bi-directional communication channel between TROMs in our implementation model. Each port has a unique *port-id* (PID). Also the link information should be kept in a port object. Figure 23 shows the class diagram of Port.

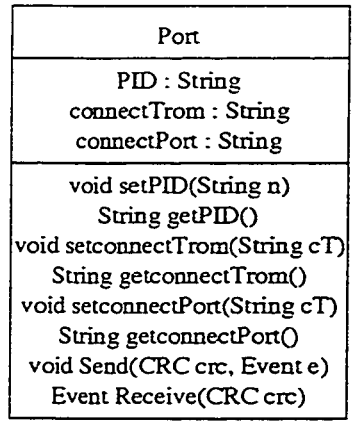


Figure 23: Class diagram of Port

The descriptions of the data members and the methods of port class are shown as

below:

- *PID* - port id.
- *connectTROM* - which TROM this port connect to.
- *connectPort* - which Port of that TROM this port connect to.
- *setPID, getPID, setconnectTrom, getconnectTrom, setconnectPort, getconnectPort* - access points for the data members.
- *send* - implementing sending an event.
- *receive* - implementing receiving an event.

Transition

A transition is a computation triggered by a specific event. Actions of a transition are the other kind of detailed actions of a reactive object. It will change some values of the attributes of the reactive object, also it will change the current state of the reactive object. Every transition has a pre-condition. A transition is triggered only the pre-condition of this transition is satisfied in our implementation model. After the actions of the transition are taken, the values of the attributes of the reactive system should satisfy the post-condition of the transition. Figure 24 shows the class diagram of Transition.

The descriptions of the data members and the methods of port class are shown as below:

- *Name:* the name of the transition.
- *sourceSN:* the name of the source state of the transition.
- *destinationSN:* the name of the destination state of the transition.
- *triggering_event:* the event which triggers the transition.
- *setName, getName, setsourceSN, getsourceSN, setdestinnationSN, getdestinnationSN, settriggeringevent, gettriggeringevent:* access points of the data members.

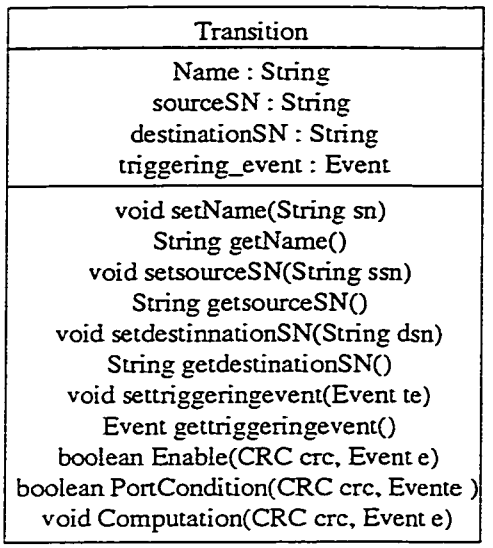


Figure 24: Class diagram of Transition

- *Enable:* a virtual function, returning true when the enable condition is satified.
- *PortCondition:* a virtual function, returning true when the port condition is satified.
- *Computation:* a virtual function, updating the values of the attributes and the current state.

Therefore, Transition class is an *abstract base class.* Every transition of a reactive object is a subclass of this class, provides the implementation of Enable, PortCondition, and Computation functions, and initializes Name, sourceSN, destinationSN and triggering_event.

Time-Constraint

A time-constraint in the implementation is a *timer.* A timer is activated by a transition. When the setting time is arrived and the TROM is in a certain state, an event should be fired. In other words, a reaction is triggered after the setting time is arrived. Therefore, if there is a time constraint in a state, the codes for checking

whether a timer arrives will be part of the actions of this state. Figure 25 shows the class diagram of TimeConstraint.

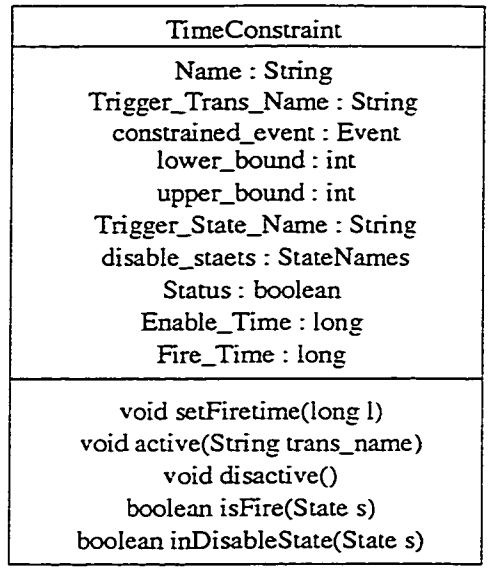


Figure 25: Class diagram of TimeConstraint

The descriptions of the data members and the methods of TimeConstraint class are shown as below:

- *Name:* the name of the time constraint.
- *Trigger_Trans_Name:* the name of the transition which activates the time constraint.
- *constrained_event:* the event that should be fired after the setting time is arrived.
- *lower_bound:* the lower time bound of the time constraint.
- *upper_bound:* the upper time bound of the time constraint.
- *Trigger_State_Name:* the name of state where the event can be fired.
- *disable_states:* the set of the disable states.
- *Status:* the status of the timeconstraint whether it is active or not.
- *Enable_time:* the time which enables the time constraint.

- *Fire_time:* the time which the event should be fired.
- *setFiretime:* sets the fire time.
- *active:* activates the time constraint.
- *disactive:* disactivates the time constraint.
- *isFire:* returns true when the constrainted event should be fired.

3.5 Composite Class

To minimize design complexity and to promote modularity at subsystem level, *composite class*, a composition at the class level, is introduced in TROM design model. Apparently, we can generate the *composing state machine* by producing the state machines of the components of the composite class. But this may not be the best way, since the work to generate the composing state machine is too complicated. For example, consider a composite class having three components, where every component has four states. The composing state machine of this composite class may have $4*4*4 = 64$ states. The more components a composite class has and the more states every component has, the more complicated the composing state machine becomes. It will not be possible to manually generate the composing state machine.

Our method to implement a composite class is based on the subsystem level implementation. Every reactive system has a super thread for task scheduling and message passing in our reactive model. Therefore, a composite class has a thread that schedules its internal reactive objects and passes messages between its internal reactive objects and the environment. To the whole reactive system, this thread is also a reactive object, and the information about the components is hidden. The time slice assigned to this thread must be bigger than (number of the components * normal time slice for a simple reactive object + communication time + schedule time). Figure 26 shows the reactive model after the refinement. The thread n in the diagram is a composite class object. The threads inside the thread n may be either a simple reactive object or a composite class object.

Using this method, the internal components of a composite class is just under the control of the thread of the composite class object, and will know nothing about

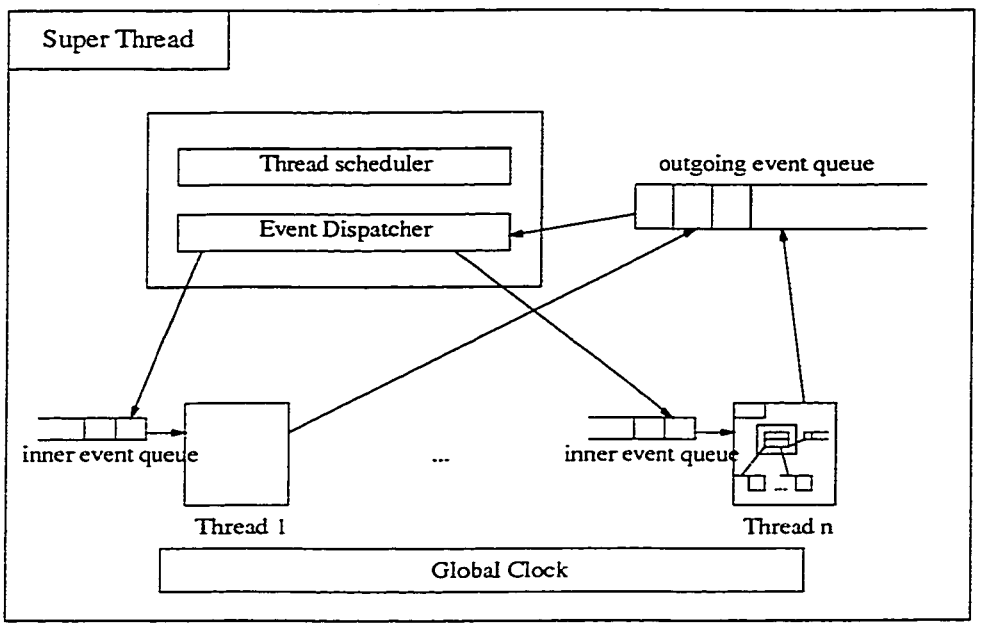


Figure 26: Refinement of Reactive Model

the whole reactive system. The whole reactive system just communications with the thread and don't need to care about the inside of this composite class object. A class named RTComposite is developed in our real-time run-time library for Java. The class has the properties of both RTSuper and RTThread.

3.6 System Implementation

A reactive system has a group of reactive objects. These objects and the interaction relationship are described as SCS in TROM design model. In the implementation, every reactive object should be created and initialized. And the interaction relationship should be provided too. Moreover, every reactive system has a super thread. Figure 79 shows the class diagram of a reactive system. *PortPara* class is for the interaction relationship between the reactive objects.

3.7 System Refinement

A system may need to be refined for some reasons. In TROM design model, only three forms of constrained inheritances based on subtyping, such as behavioral inheritance,

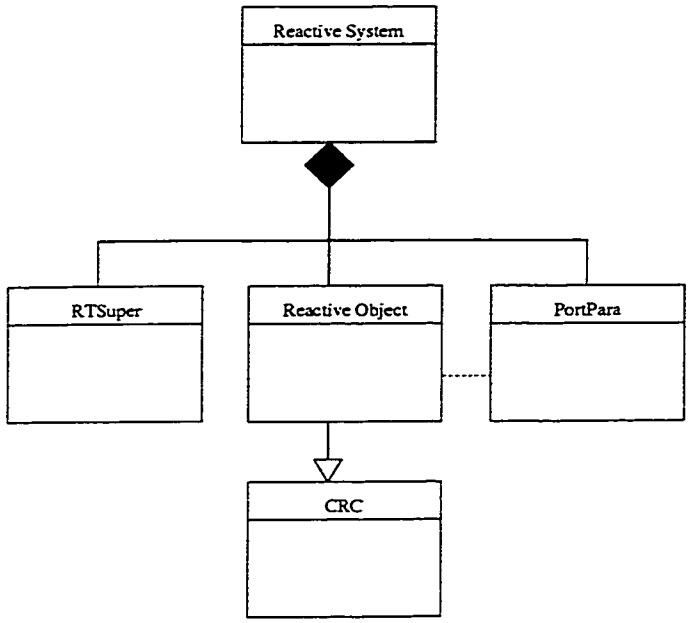


Figure 27: Class Diagram of Reactive Object

extensional inheritance, and polymorphic inheritance, are allowed. Therefore, the new reactive object class must be a subclass of the original reactive object class.

The major behaviors of the design refinement are:

- Attribute redefinition - The data model of an attribute is redefined. In other words, the data type of an attribute can be changed.
- Transition redefinition - The post-condition, port-condition, and enabling-condition of a transition is strengthened.
- Time-constraint redefinition - The minimal time daly is increased or the maximal time delay may be decreased.
- Event addition - New events are added.
- Port addition - New port-types are added.
- State addition - New states are added.
- Attribute addition - New attributes are added.
- Transition addition - New transitions are added.

- Time-constraint addition - New time-constraints are added.

These behaviors makes the final implementation to change. In our implementation model, abstract data types, such as Ports, State, Transitions, and TimeConstraints, are introduced. Obviously, port addition, state addition, transition addition, and time-constraint addition mean adding an element into a set. The functions for adding a new element are already provided. So we do not need to do anything for these four behaviors. Event is not an independent element in our implementation model. Adding a new event implies a transition is added. Therefore, a transition addition implements an event addition. Transition redefinition and time-constraint redefinition mean an old transition or an old time-constraint should be replaced with a new one. So, the functions for replacing a transition and a time-constraint are provided in abstract data types, Transitions and TimeConstraints. Attributes are defined as data members of a reactive object. So attribute redefinition means an attribute should be overridden with a new data type in the new reactive object class. Attribute addition means a new attribute should be introduced in the new reactive object class.

Chapter 4

Automated Code Generation

4.1 Introduction

In Chapter 3, an implementation model for TROM design model is presented. In this chapter, we deal with the problem of automatic code generation, which means automating the translation of formal specifications into an implementation for a desired target environment. In this thesis, the target environment is *Real-Time Java* (RT-Java), which is Java platform plus the real-time run-time library for Java. TROM support library and Abstract Data Type libraries are also parts of the environment.

Automatic code generation is one of the most promising concepts in the software development process, because it allows for design models to retain their usefulness throughout the product's life-cycle. However, automatic code generation is not a deeply researched area. There does not exist a general strategy developed in this area.

In our case, the problem of automatic code generation can be restated as automatically merging the implementation model and generating the detail actions, such as actions of transitions and actions of states. The design and the implementation details of the automatic code generation are presented in later sections.

4.2 Description of the AST

Internally, TROM specifications are represented by an *abstract syntax tree (AST).* This specification was syntactically checked as the AST is constructed. The AST, as its name implies, is a tree of links of all components of TROM classes and subsystems with access methods to all of these components. Therefore, instead generating code from formal specifications directly, codes can be generated from AST. Figure 28 shows the high level structure of the AST with the components used by the generator developed.

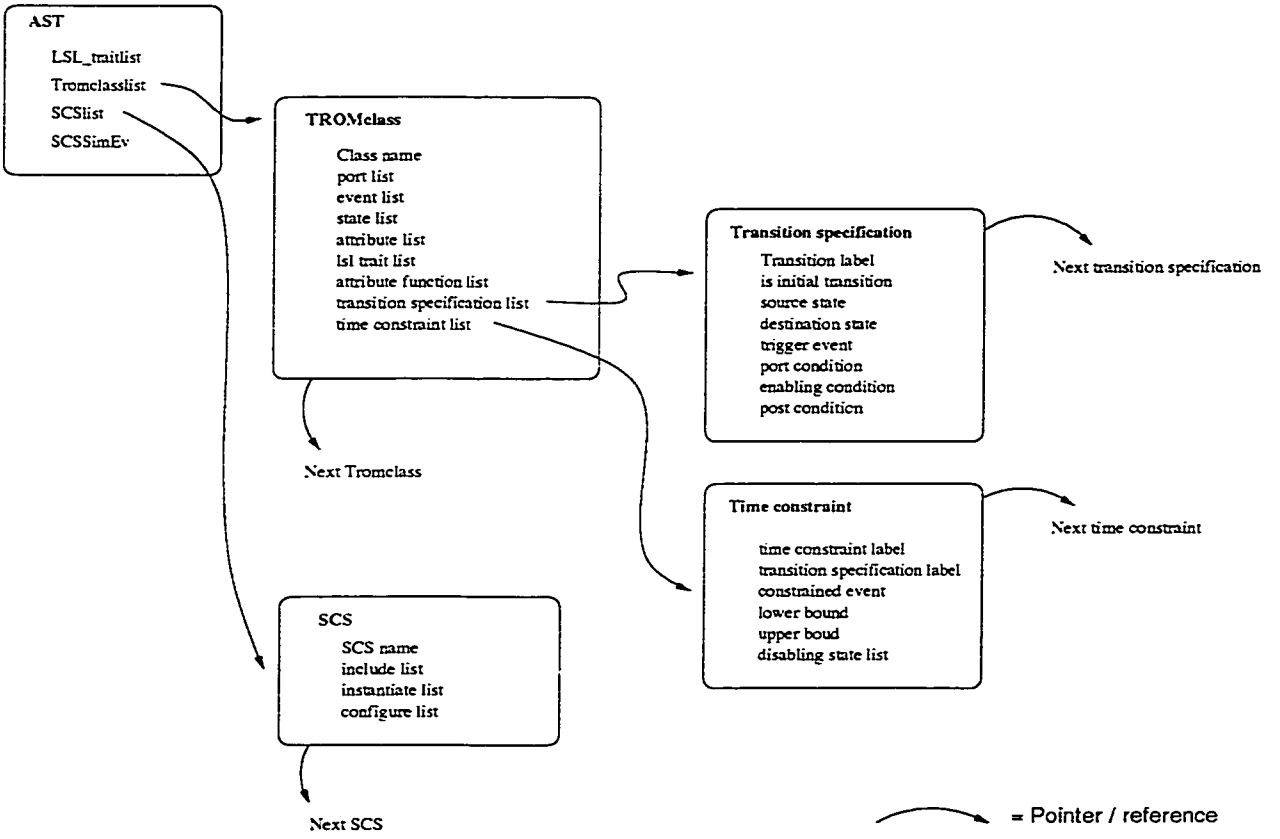


Figure 28: High level AST structure with subset of components shown

The development of the interpreter to construct the AST was originally done in the C++ environment [Tao96] and then ported to Java [Sri99]. The code generator tool was done with the latest version.

4.3 Implementing Abstract Syntax Tree of LIL

The Larch Interface Language (LIL) is quoted for increasing the reusability of abstract data types and decreasing the association between Abstract Data Type Implementation and TROM Implementation Model in our implementation methodology. Furthermore, if the assumption that every signature in LSL trait has an independent function to implement it is accepted, automatic code generation for transitions are possible since every transition is described by using the assertions between attributes or using function signatures among abstract data types in Larch.

Based on this assumption, we can extract the information about the corresponding relationships between functions and signatures from LIL specifications. In our case, Java is chosen as the programming language. So, Larch/Java is chosen as the Larch Interface Language. However, the interface language Larch/Java has not been studied. Based upon the relationship between C++ and Java, we have designed the syntax and semantics of Larch/Java language from Larch/C++ [LEA99].

Figure 29 shows a Larch/Java specification for Set. The syntax of Larch/Java consists of two parts. The first part borrows from Java for declaring class name and functions. The second part borrows from Larch/C++ for describing the behaviors of functions. You may notice that all sentences of the second part begins with "//@" in Figure 29. Adding this identity before the behavior specifications makes the implemented code and the behavior specifications to exist in the same file since the sentences beginning with "//" are comments in Java. Therefore, the reusability of abstract data types is increased.

Our syntax parser and checker for Larch/Java will ignore all of the sentences except the declarations for class name, functions, and the sentences begin with "//@". The internal representation of a Larch/Java specification is similar to the AST of TROM. Figure 30 shows the high level structure of the AST of Larch/Java with the components used by the code generator.

```
public class Set
{
   //@ uses Set(Set for S, Object for E);

   Set()
   {
      //@ modifies self;
      //@ ensures self' = create();
   }

   public void add(Object o)
   {
      //@ modifies self;
      //@ ensures self' = insert(o, self^);
   }

   public void remove(Object o)
   {
      //@ requires member(o, self^);
      //@ modifies self;
      //@ ensures self' = delete(o, self^);
   }

   public int size()
   {
      //@ ensures result=size(self^);
   }

   public boolean member(Object o)
   {
      //@ ensures result=member(o, self^);
   }

   public boolean isEmpty()
   {
      //@ ensures result = isEmpty(self^);
   }
}
```

Figure 29: Larch/Java specification for Set

4.4 From TROM To Code

In Chapter 3, the implementation model of the TROM has been described. Based on this model, it is quite possible to automatically map TROM specifications into an implementation. In this section the mechanization of automatic code generation is explained.

Reactive objects have similar high level behavior, however they differ in their detailed behaviors, such as actions of transitions and actions of states. Therefore, the code generation consists of four parts, such as transition class generation, state class generation, reactive class generation, and system class generation. Figure 31 shows the algorithm of the code generation algorithm.

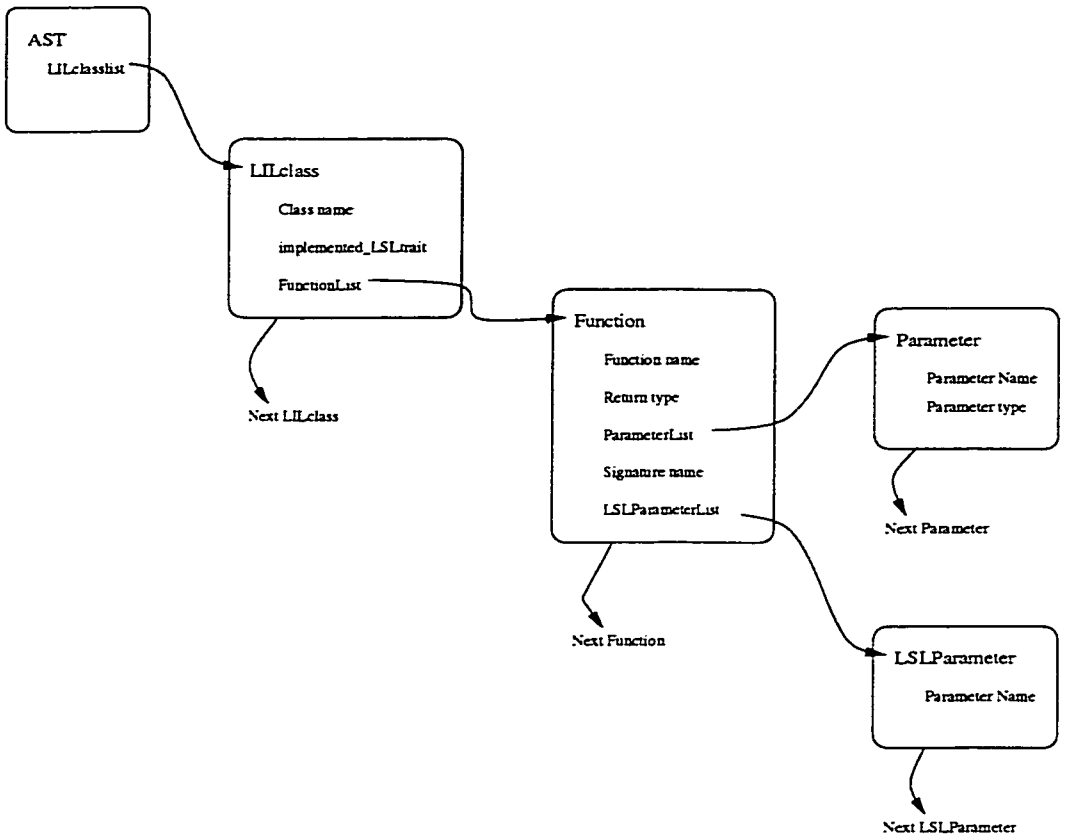


Figure 30: High level AST structure of LIL

4.4.1 Transition Class Generation

Transition classes are subclasses of the class *Transition*. The subclasses have more detailed information that are necessary for implementing transition specifications. We generate *Transition classes* by extracting appropriate information of the transition specifications in a TROM specification. The name of transition, the source and destination states, and triggering event labeling the transition are easy to extract from a transition specification. The predicates in a post-condition implicitly describe the action associated with a transition. Program code for an action is an operational translation of these predicates. Moreover, the port-condition and the enabling condition from a transition are translated to provide the choice points in the program description. Figure 28 shows the algorithm for generating a *Transition class*.

A predicate in TROM is an assertion either between attributes or between signatures used among abstract data types. Generally, an assertion between attributes is

```
create AST for TROM
create AST for LIL

for each transition in the AST do
    generate the transition class for this transition
end for

for each state in the AST do
    generate the state class for this state
end for

for each TROM in the AST do
    generate the reactive class for this TROM
end for

generate the system class
```

Figure 31: Pseudo-code for the whole code generation

```
generate the class declaration
generate the constructor function
generate the Portcondition function
generate the Enable function
generate the Computation function
generate the end of the class
```

Figure 32: Pseudo-code for every transition class generation

a comparison between attributes. Therefore, transforming this kind of assertions is to replace attributes in TROM specification with attributes defined in TROM implementation. Figure 33 shows an example of the translation of this kind of assertions. Although the two predicates have the same operator "=", the generated codes are different since the semantics of pre-condition and post-condition are different.

Translating an assertion using signatures among abstract data types is more complicated. First of all, we have to find out the function that corresponds to the signature used in the predicate. Based on the assumption that every signature has an independent function for implementing it, we can search the signature in the AST of LIL to find out the corresponding function. Second, we have to translate the signature to a function call. The relationships between parameters of the function and parameters

```
Port Condition : cr = pid       ===>   ((Train)crc).cr.equals(((Train)crc).PID)
Post Condition : cr' = pid      ===>   ((Train)crc).cr = ((Train)crc).PID;

                 Predicates            Java Codes
```

Figure 33: Translating assertions between attributes to codes

of the signature must be found out. Figure 34 shows the pseudo code of the algorithm.

```
search the function which corresponds to the signature
get the parameter list of the function
get the parameter list of the signature

for each parameter in the parameter list of the signature do
    search this parameter in the parameter list of the function
    if this parameter can not be found
          in the parameter list of the function then
       generate the object's name
       output ``.''
    end if
end while

output the name of the function
output ``(''

for each parameter in the parameter list of the function do
    search this parameter in the parameter list of the signature
    generate the parameter

    if not the last parameter in the parameter list of the function then
       output ``,''
    endif
endwhile

output ``)''
```

Figure 34: Pseudo-code for translating a signature to a function call

A pre-condition of a transition and a post-condition of a transition may be a conjunction of a group of predicates. For a pre-condition, it may not change any attributes of a reactive object, so the executed order of predicates is not relevant. But for a post-condition, the executed order must be considered. For example, consider a post-condition $A' = head(Q) \wedge Q' = tail(Q)$, where Q is a queue. The signature *head* gets the first element of the queue, and the signature *tail* removes the first element from

the queue. Assume function *head* and *tail* correspond to the signatures *head* and *tail*. $A' = head(Q)$ can be translated to $A = Q.head()$, and $Q' = tail(Q)$ can be translated to $Q.tail()$. Therefore, we must assure $A = Q.head()$ is executed before $Q.tail()$. So we must assure $A' = head(Q)$ is generated first in the code generation. An algorithm is provided to solve this problem. Figure 35 shows the pseudo code of the algorithm.

```
create a empty queue named pre-queue
for each predicate
    find out which attributes are used in the predicate
         and which attributes are modified in the predicate

    put the predicate plus these information into pre-queue
end for

create a empty queue named post-queue
while pre-queue is not empty
      get the first predicate of pre-queue
      if the attributes that will be modified in this predicate
            are not used in other predicates
         put this predicate into post-queue
      else put this predicate into pre-queue again.
end while
/* the predicates are ordered and are put in post-queue*/
```

Figure 35: Pseudo-code for the algorithm to decide the executed order

In the algorithm, we determine the attributes that are used but not modified in a predicate and the set of attributes that are modified by the predicate. The post-condition can be a single predicate or a conjunction of several predicates; it cannot contain a disjunction. For example, if the post-condition is of the form $A' = 1 \vee A' = 2$, the action is non-deterministic and consequently no code for a deterministic computation can be generated. However, both conjunctions and disjunctions may be present in the enabling condition and the port-condition. Another important requirement for mechanical code generation is that the predicates explicitly provide sufficient information for an implementation. As an example, the predicate $inSet' = insert(pid, inSet)$ explicitly states the modification of the attribute *inSet* due to the invocation of the operation corresponding to the abstract term *insert*, whereas the predicate $member(pid, inSet')$ implicitly asserts the same action, but provides no clue as to the operation that effects the modification. The mechanical code generator works

without assistance when explicit specifications are given. Therefore, the grammar of an acceptable post-condition for automatic code generation is defined in Table 1. The attribute on the left side of "=" is the attribute that will be modified. The attributes on the right side of "=" are the attributes that will be used.

post-condition	::=	<predicate> { <l_op> <predicate> }
l_op	::=	<AND>
predicate	::=	<att_name>"'" <b_op> <factor>
b_op	::=	=
factor	::=	<att_name> | <LSL_term>
LSL_term	::=	<LSL_func_name>"("<arg_list>")"
arg_list	::=	<arg> { "," <arg> }
arg	::=	"pid" | <att_name> | <LSL_term>
att_name	::=	String
AND	::=	"&"

Table 1: Grammar of an acceptable post-condition

4.4.2 State Class Generation

State classes are subclasses of the class *State*. The subclasses provide sufficient information for implementing a state. States are considered as choicepoints in our implementation model. Therefore, actions of a state are to decide which state is the next state. The behavior of waiting for an event from the environment is already abstracted as the high level behavior of a reactive object. The behaviors of checking some conditions and then firing an internal/outgoing event are actions of a state. So generating code for the actions of a state is to find out which events should be fired in this state and under which condition the event should be fired.

The information about which events should be fired/received in a state can be attained from transition specifications in TROM design model. By checking the type of events, the incoming events can be removed. An internal/outgoing event may be either a time-constraint event or an unconstraint event. A time-constraint event means that the event occurs only when the setting time is arrived. An unconstraint event means that the event occurs as soon as the enabling condition of the transition triggered by

this event is satisfied. Figure 36 shows the algorithm of state class generation.

```
generate the class declaration
generate the constructor function
generate the declaration of the run function

for each transition do
    if the source state of this transition = this state then
       if the type of the triggering event of this transition
             is not incoming then
          found = false
          for each time-constraint do
              if the constraint event of this time-constraint
                    = the triggering event then
                 generate codes for the time-constraint event
                 found = true
              end if
          end for

          if found = false then
             generate codes for the unconstraint event
          end if
       end if
    end if
end for

generate the end of the run function
generate the end of the class
```

Figure 36: Pseudo-code for the algorithm of state class generation

4.4.3 Reactive Class Generation

A reactive class is a subclass of *Common Reactive Class (CRC)*. Generating a reactive class is to translate attributes in TROM design model into data members of this reactive class and generate the codes for adding ports, states, transitions, and time-constraints into this reactive class. So, the algorithm is simple and is shown in Figure 37.

```
generate the class declaration

for each attribute in this TROM specification do
    generate a data member from this attribute
end for

generate the declaration of the constructor function

for each attribute in this TROM specification do
    generate codes for initializing the data member
end for

for each port in this TROM specification do
    generate codes for adding this port
end for

for each state in this TROM specification do
    generate codes for adding this state
end for

for each transition in this TROM specification do
    generate codes for adding this transition
end for

for each time-constraints in this TROM specification do
    generate codes for adding this time-constraint
end for

generate end of the constructor function
generate end of the class
```

Figure 37: Pseudo-code for the algorithm of reactive class generation

4.4.4 System Class Generation

A reactive system consists of a group of reactive objects. The code of a system class defines the connections between reactive objects and initializes these reactive objects. A system class can be generated from a SCS specification since all the necessary information is in the SCS specification. The algorithm is shown in Figure 38.

```
generate the class declaration
generate the declaration of the main function
for each TROM object in SCS do
    generate codes for defining the connections between reactive objects
    generate codes for creating this object
end for
generate codes for starting this system
generate the end of the main function
generate the end of the class
```

Figure 38: Pseudo-code for the algorithm of system class generation

Chapter 5

Case Study: The Railroad Crossing Problem

5.1 Introduction

This chapter will demonstrate our implementation methodology and the application of the automatic code generation. We consider an example of a railroad crossing problem. This problem has been defined as a benchmark problem by the real-time reactive system community and has been discussed previously in [Ach95], and [AM98] as a case study to illustrate the expressivity of the TROM formalism.

In this section, the example will first be outlined informally followed by its formal descriptions in the TROM notation. Then our implementation methodology will be applied to implement this example. Finally we will comment on the discrepancies between the automatically generated codes and the manually generated codes and provide some justification.

5.2 The Railroad Crossing Problem

5.2.1 An Informal Description

A railroad crossing system consists of a collection of trains and a collection of gates servicing the roads crossing the train tracks. The gate should remain closed whenever a train goes past the crossing. In order to control the gates there exists a collection

of controllers such that one controller controls each gate. A controller closes its associated gate when it gets a "nearing signal" from a train and opens the gate once all the trains crossing the gate have left. A controller does this by receiving signals from the trains and transmitting necessary control signals to its associated gate. Furthermore, more than one train can cross a gate simultaneously, through multiple parallel tracks; a train can independently choose the gate it will cross, probably based on its destination. The entities interacting in the system are trains, controllers, and gates. The safety property is that the controller is active and the gate remains closed until all the trains crossing it leave the gate.

A train sends a *Near* message to a controller indicating that it is approaching the gate associated with this controller. While leaving the gate, the train informs the controller by an *Exit* message. A typical time constraint on the train is that the *Exit* message must be sent within a window of 2 to 6 time units after sending the *Near* message.

A controller, upon receiving a *Near* message from a train, sends the *Lower* message to the gate it is controlling, indicating that the gate has to be lowered. The controller keeps monitoring trains entry and leave the gate until receives an *Exit* from the last train to leave the gate. Then the controller sends a *Raise* message to the gate. There are two time constraints associated with the controller. The controller should respond by sending: i) a *Lower* message to the gate within 1 time unit after receiving the *Near* message from the first train to entry the gate; ii) a *Raise* message to the gate within 1 time unit after receiving the *Exit* message from the last train to leave the gate.

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

5.2.2 Train-Gate-Controller Model

There are three types of interacting entities: *Train*, *Gate* and *Controller*. The behavior of the systems components are modeled using generic reactive classes as shown in Figure 39. Each of the GRC classes has a UML statechart diagram for describing the behavior of this GRC.

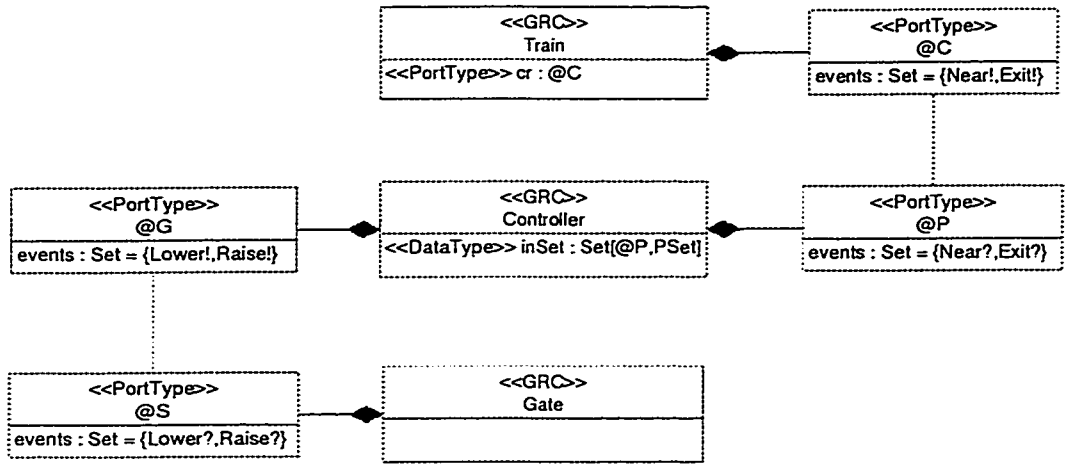


Figure 39: Train-Gate-Controler System Class Diagram

Figure 40 and 41 show the statechart diagram and the formal specification for *Train*. Initially, a *Train* is in state *Idle*. When the train approaches a *Gate*, event *Near* is sent to the *controller* associated with the gate. The train goes into state *toCross*. The attribute *cr* denotes that further interaction of the train should be at the port it chose, until it exits the gate. Within 2 to 4 time units the train enters the gate. An internal event *In* is sent to signify the start of the action of crossing the gate. The train enters into state *Cross*. Also an internal event *Out* is sent at the end of the action of crossing the gate. The train goes into state *Leave*. Then event *Exit* is sent to the controller. The train is idle again.

In the beginning, the controller is in state *idle*. When the input event *Near* from the train is received, the controller goes into state *active*. The attribute *inSet* is introduced for denoting the set of ports at which an interaction involving *Near* has occurred, and implicitly underscores those train objects crossing the gate at that instant. The controller sends the event *Lower* to the gate within 1 time unit, then enters into state *monitor*. In state *monitor*, the controller deals with the trains' enters and exits. Until the last train in the gate leaves, the controller goes into state *deactive*. With 1 time unit, the controller sends the event *Raise* to the gate. Figure 42 and 43 show the statechart diagram and the formal specification for *Controller*.

Figure 44 and 45 show the statechart diagram and the formal specification for *Gate*. A gate is opened in state *opened* and closed in state *closed*. When the gate receives event *Lower/Raise*, the action to close/open the gate is taken. The events *Down* and

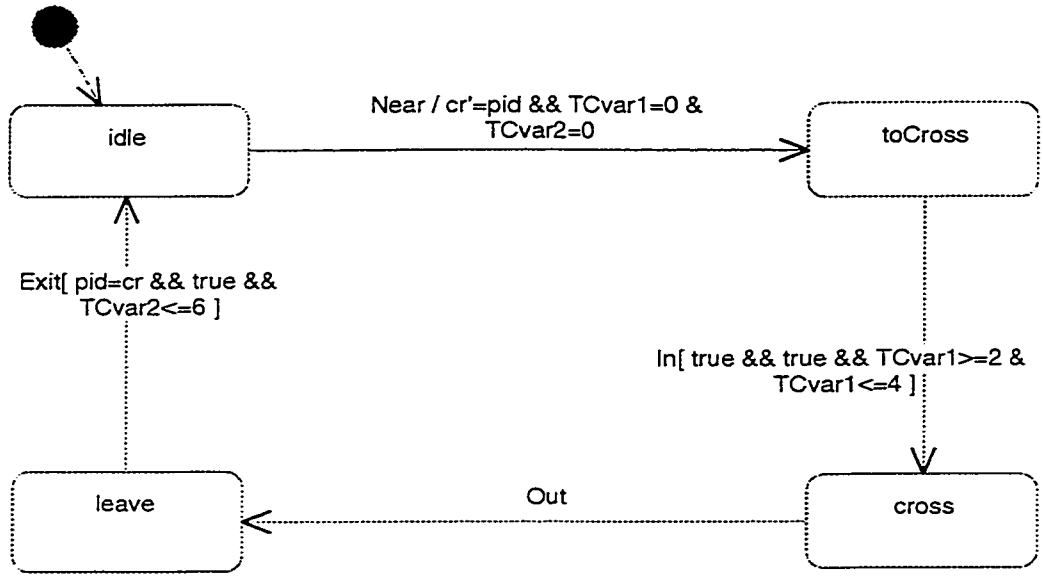


Figure 40: Statechart Diagram for Train

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: <toCross,cross>; In(true); true ⟹ true;
R3: <cross,leave>; Out(true); true ⟹ true;
R4: <leave,idle>; Exit(pid=cr); true ⟹ true;
Time-Constraints:
TC2: R1, Exit, [0, 6], {};
TC1: R1, In, [2, 4], {};
end

Figure 41: Formal specification for GRC Train

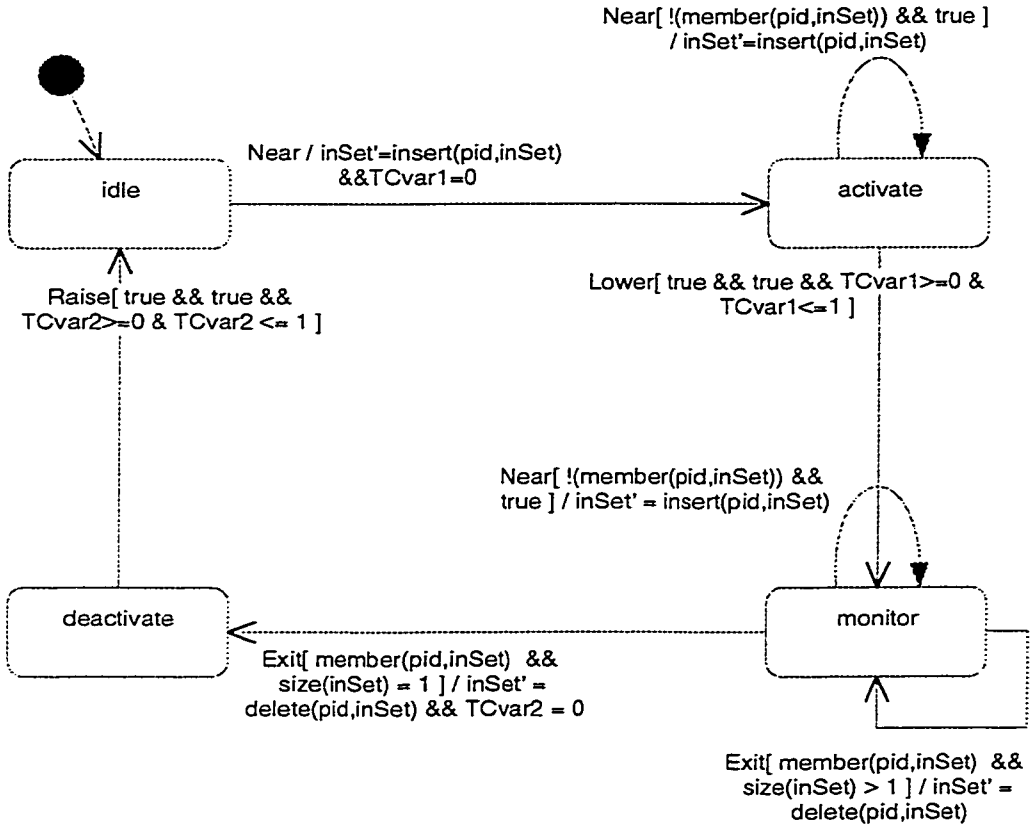


Figure 42: Statechart Diagram for Controller

```
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:
     TC1: R7, Lower, [0, 1], {};
     TC2: R4, Raise, [0, 1], {};
end
```

Figure 43: Formal specification for GRC Controller

Up denote the end of the actions.

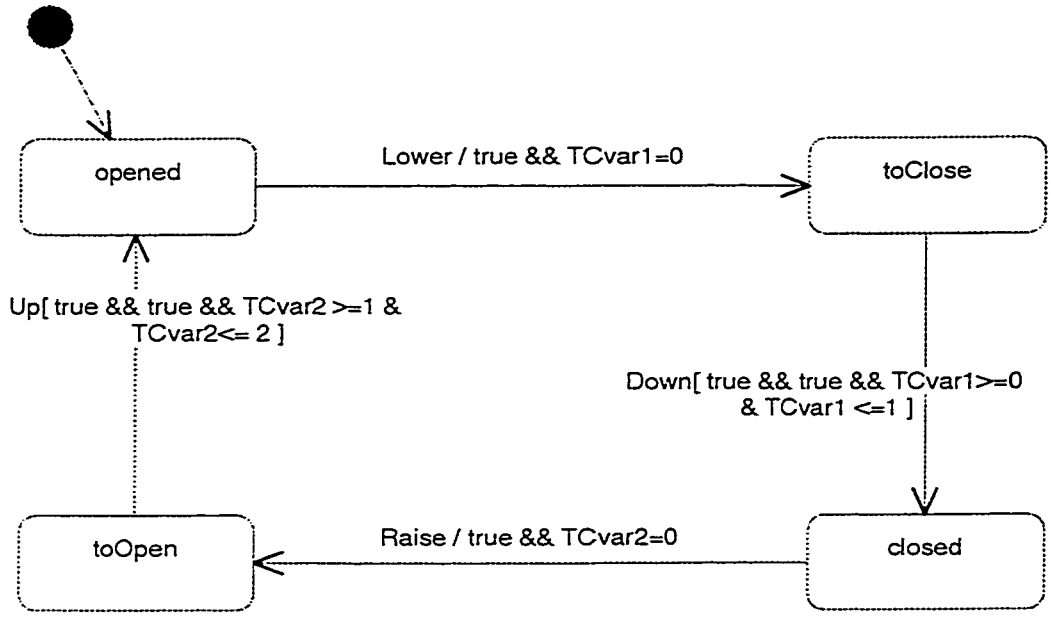


Figure 44: Statechart Diagram for Gate

The Railroad subsystem which has 5 trains, 2 controllers, and 2 gates is the example of reactive systems considered for mechanical code generation. Figure 46 shows the collaboration diagram for the subsystem. Figure 47 shows the formal notation for the subsystem.

5.3 Implementation of Train-Gate-Controller

The code generation algorithm discussed in Chapter 4 is applied to the Train-Gate-Controller design specifications. The code generator by the algorithm very nearly matches the manual code.

5.3.1 Abstract Data Type Classes

In Train-Gate-Controller model, only one abstract data type *Set* is introduced. So our work of this tier is to implement the class *Set*, using the interface and behavior specifications of *Set* is shown in Figure 29.

Class Gate [@S]
Events: Lower?@S, Down, Up, Raise?@S
States: *opened, toClose, toOpen, closed
Attributes:
Traits:
Attribute-Function: opened → {}; toClose → {}; toOpen → {}; closed → {};
Transition-Specifications:
R1: <opened,toClose>; Lower(true); true ⟹ true;
R2: <toClose,closed>; Down(true); true ⟹ true;
R3: <closed,toOpen>; Raise(true); true ⟹ true;
R4: <toOpen,opened>; Up(true); true ⟹ true;
Time-Constraints:
TC1: R1, Down, [0, 1], {};
TC2: R3, Up, [1, 2], {};
end

Figure 45: Formal specification for GRC Gate

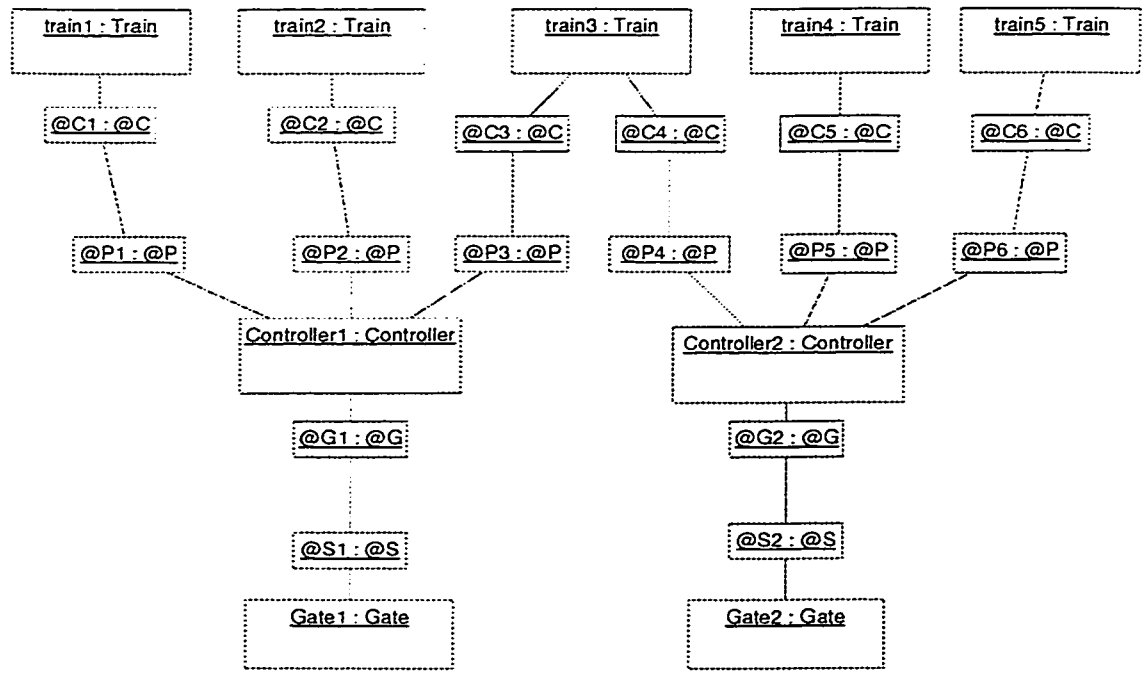


Figure 46: Collaboration diagram for subsystem TrainGateController

```
SCS TGC
    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 47: Formal specification for subsystem TrainGateController

5.3.2 Reactive Object Classes

In Train-Gate-Controller model, there are three types of reactive objects: *Train*, *Gate*, and *Controller*. We discuss the implementations of these three types of reactive objects in the later of this section.

5.3.2.1 Train

First of all, we implement every transition in Train. Then the implementations of the states in Train are provided. Finally, the train class is given.

Transition There are four transitions in Train.

1. Transition R1: The port condition and enabling condition are *true*. So Function *PortCondition* and *Enable* just return true. The post condition is $cr' = pid$. So the attribute *cr* is changed in Function *Computation*. Also the function should change the current state to *toCross*. Figure 48 shows the implementation of R1.
2. Transition R2: The port condition, enabling condition, and post condition are *true*. So Function *PortCondition* and *Enable* just return true. Function *Computation* change the current state to *cross*. Figure 49 shows the implementation of R2.
3. Transition R3: Function *PortCondition* and *Enable* return *true*. Function *Computation* changes the current state to *leave*. Figure 50 shows the implementation of R3.
4. Transition R4: Function *PortCondition* returns *true* under *pid* is equal to *cr*. Function *Enable* returns *true*. Function *Computation* changes the current state to *idle*. Figure 51 shows the implementation of R4.

State There are four states in Train.

1. State *idle*: There is only an outgoing event *Near* that can happen in this state. This event is an unconstraint outgoing event. So the event can only happen at a port that satisfies the port condition of the transition R1 when the enabling condition of R1 is satisfied. Unfortunately, the specification

```
public class Train_R1 extends Transition
{
   Train_R1()
   {
      super("R1", "idle", "toCross", new Event("Near"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      return(true);
   }

   public void Computation(CRC crc, Event e)
   {

      ((Train)crc).cr = crc.PID;

      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 48: Implementation for R1 of Train

```
public class Train_R2 extends Transition
{
   Train_R2()
   {
      super("R2", "toCross", "cross", new Event("In"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      return(true);
   }

   public void Computation(CRC crc, Event e)
   {
      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 49: Implementation for R2 of Train

```
public class Train_R3 extends Transition
{
   Train_R3()
   {
      super("R3", "cross", "leave", new Event("Out"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      return(true);
   }

   public void Computation(CRC crc, Event e)
   {
      crc.getState().setcState(sourceSN,destinationSN);
   }
}
```

Figure 50: Implementation for R3 of Train

```
public class Train_R4 extends Transition
{
   Train_R4()
   {
      super("R4", "leave", "idle", new Event("Exit"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      if (((Train)crc).cr.equals(((Train)crc).PID))
         return(true);

      return(false);
   }

   public void Computation(CRC crc, Event e)
   {
      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 51: Implementation for R4 of Train

Train lacks of detailed information under which the event is fired in a specific port. We refine the transition specification by introducing a new abstract data type *Sensor*, which has two functions *trigger* and *selectPort*. Function trigger returns true when the event should be fired. Function selectPort returns a port id where the event is fired. A new attribute *q*, which is a *Sensor*, is introduced in Train. Then the transition R1 becomes:

<idle,toCross>; Near(pid=selectPort(q)); trigger(q) $\Longrightarrow$ cr'=pid;

In other words, the original R1 means the event *Near* will be broadcasted at all of the ports after the train enters state *idle*. Figure 52 shows the refined Train specification and Figure 53 shows the implementation of the state *idle* with the new design.

2. State *toCross*: There is only an internal event *In* that can happen in this state. This event is a time-constraint event. Therefore, the event is fired at the *null* port when the setting time is arrived. Figure 54 shows the implementation of state *toCross*.
3. State *cross*: There is only an internal event *Out* that can happen in this state. This event is a non-time-constraint event. Figure 55 shows the implementation of state *cross*.
4. State *leave*: There is only an outgoing event *Exit* that can happen in this state. This event is a time-constraint event. Therefore, the event is fired when the setting time is arrived. Figure 56 shows the implementation of state *leave*.

Reactive Class In Train, there are one attribute, one port-type, four states, four transitions and two time-constraints. Figure 57 shows the implementation of Train.

5.3.2.2 Gate

Transition There are four transitions in Gate.

1. Transition R1: Function *PortCondition* and *Enable* return *true*. Function em Computation changes the current state to *toClose*. Figure 58 shows the implementation of R1.

Class Train [@C]
Events: Near!@C, Out, Exit!@C, In
States: *idle, cross, leave, toCross
Attributes: cr:@C, q: Sensor
Traits: Sensor
Attribute-Function: idle → {}; cross → {} ;leave → {}; toCross → {cr};
Transition-Specifications:
R1: <idle,toCross>; Near(pid=selectPort(q)); trigger(q) ⟹ cr'=pid;
R2: <toCross,cross>; In(true); true ⟹ true;
R3: <cross,leave>; Out(true); true ⟹ true;
R4: <leave,idle>; Exit(pid=cr); true ⟹ true;
Time-Constraints:
TC2: R1, Exit, [0, 6], {};
TC1: R1, In, [2, 4], {};
end

Figure 52: Refinement for GRC Train

```
class Train_idle extends State
{
   Train_idle()
   {
      super("idle", true, new States());
   }

   public void run(CRC crc)
   {
      if ((Train)crc).q.trigger()
      {
         Port p = ((Train)crc).q.selectPort();

         p.Send(crc, new Event("Near"));
      }
   }
}
```

Figure 53: Implementation for State *idle* of Train

```
class Train_toCross extends State
{
   Train_toCross()
   {
      super("toCross", false, new States());
   }

   public void run(CRC crc)
   {
      int j =  crc.getTimeConstraints.searchTimeConstraint("TC1");

      if(crc.getTimeConstraints.TimeConstraintAt(j).isFire())
      {
         int i = crc.getPorts().searchPort("null");

         crc.getPorts().portAt(i).Send(crc, new Event("In"));
         crc.getTimeConstraints.TimeConstraintAt(j).disactive();
      }
   }
}
```

Figure 54: Implementation for State *toCross* of Train

```
class Train_cross extends State
{
   Train_cross()
   {
      super("cross", false, new States());
   }

   public void run(CRC crc)
   {
      int i = crc.getPorts().searchPort("null");

      crc.getPorts().portAt(i).Send(crc, new Event("Out", "Internal", "null" ));
   }
}
```

Figure 55: Implementation for State *cross* of Train

```
class Train_leave extends State
{
   Train_leave()
   {
      super("leave", false, new States());
   }

   public void run(CRC crc)
   {
      int j =  crc.getTimeConstraints.searchTimeConstraint("TC1");

      if(crc.getTimeConstraints.TimeConstraintAt(j).isFire())
      {
         ((Train)crc).cr.Send(crc, new Event("Exit", "GC", "Outgoing"));
         crc.getTimeConstraints.TimeConstraintAt(j).disactive();
      }
   }
}
```

Figure 56: Implementation for State *leave* of Train

2. Transition R2: Function *PortCondition* and *Enable* return *true*. Function em Computation changes the current state to *closed*. Figure 59 shows the implementation of R2.
3. Transition R3: Function *PortCondition* and *Enable* return *true*. Function em Computation changes the current state to *toOpen*. Figure 60 shows the implementation of R3.
4. Transition R4: Function *PortCondition* and *Enable* return *true*. Function em Computation changes the current state to *opened*. Figure 61 shows the implementation of R4.

State There are four states in Gate.

1. State *opened*: In this state, the gate just waits for an event *Lower* from the environment. So there is no action in this state. Figure 62 shows the implementation of state *opened*.
2. State *toClose*: There is only an internal event *down* that can happen in this state. This event is a time-constraint event. Therefore, the event is fired when the setting time is arrived. Figure 63 shows the implementation of state *toClose*.

```
class Train extends CRC
{
   Port cr;
   Sensor q;

   Train(String n, PortParas pps, Supervisor sp)
   {
      super(n, sp);

      cr = null;
      q = new Sensor();

      PortPara pp;
      pp = (PortPara)pps.elementAt(0);
      for (int i = 0; i < pp.getPortNum(); i ++)
         addPort(new Port(pp.getportID(i), pp.getConnectTrom(i), pp.getConnectPort(i)));
      addPort(new Port("null", Name, "null"));

      addState("main", new Train_idle());
      addState("main", new Train_toCross());
      addState("main", new Train_cross());
      addState("main", new Train_leave());
      mainState.initialcState();

      addTransition(new Train_R1());
      addTransition(new Train_R2());
      addTransition(new Train_R3());
      addTransition(new Train_R4());

      addTimeConstraint(new TimeConstraint(
            "TC1", "R1", new Event("In"), 2, 4, "toCross", new StateNames());
      addTimeConstraint(new TimeConstraint(
            "TC2", "R1", new Event("Exit"), 0, 6, "leave", new StateNames());
   }

   public void display_attributes(PrintStream out)
   {
      out.println(" There is 1 attribute in this TROM object.\n");
      out.println("    1th Attribute Name :  cr");
      out.println("        Attribute Value: " + cr);
   }
}
```

Figure 57: Implementation for Train

```
public class Gate_R1 extends Transition
{
   Gate_R1()
   {
      super("R1", "opened", "toClose", new Event("Lower"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      return(true);
   }

   public void Computation(CRC crc, Event e)
   {
      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 58: Implementation for R1 of Gate

3. State *closed*: In this state, the gate just waits for an event *Raise* from the environment. So there is no action in this state. Figure 64 shows the implementation of state *closed*.

4. State *toOpen*: There is only an internal event *up* that can happen in this state. This event is a time-constraint event. Therefore, the event is fired when the setting time is arrived. Figure 65 shows the implementation of state *toOpen*.

Reactive Class In Gate, there are one port-type, four states, four transitions and two time-constraints. Figure 66 shows the implementation of Gate.

5.3.2.3 Controller

Transition There are seven transitions in Controller.

1. Transition R1: Function *PortCondition* and *Enable* just return true. Function *Computation* changes the current state to *monitor*. Figure 67 shows the implementation of R1.

```
public class Gate_R2 extends Transition
{
   Gate_R2()
   {
      super("R2", "toClose", "closed", new Event("Down"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      return(true);
   }

   public void Computation(CRC crc, Event e)
   {
      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 59: Implementation for R2 of Gate

```
public class Gate_R3 extends Transition
{
   Gate_R3()
   {
      super("R3", "closed", "toOpen", new Event("Raise"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      return(true);
   }

   public void Computation(CRC crc, Event e)
   {
      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 60: Implementation for R3 of Gate

```
public class Gate_R4 extends Transition
{
   Gate_R4()
   {
      super("R4", "toOpen", "opened", new Event("Up"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      return(true);
   }

   public void Computation(CRC crc, Event e)
   {
      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 61: Implementation for R4 of Gate

```
class Gate_opened extends State
{
   Gate_opened()
   {
      super("opened", true, new States());
   }

   public void run(CRC crc)
   {
   }
}
```

Figure 62: Implementation for State *opened* of Gate

```
class Gate_toClose extends State
{
   Gate_toClose()
   {
      super("toClose", false, new States());
   }

   public void run(CRC crc)
   {
      int j = crc.getTimeConstraints.searchTimeConstraint("TC1");

      if(crc.getTimeConstraints.TimeConstraintAt(j).isFire())
      {
         int i = crc.getPorts().searchPort("null");

         crc.getPorts().portAt(i).Send(crc, new Event("Down", "Internal", "null"));
         crc.getTimeConstraints.TimeConstraintAt(j).disactive();
      }
   }
}
```

Figure 63: Implementation for State *toClose* of Gate

```
class Gate_closed extends State
{
   Gate_closed()
   {
      super("closed", false, new States());
   }

   public void run(CRC crc)
   {
   }
}
```

Figure 64: Implementation for State *closed* of Gate

```
class Gate_toOpen extends State
{
   Gate_toOpen()
   {
      super("toOpen", false, new States());
   }

   public void run(CRC crc)
   {
      int j = crc.getTimeConstraints.searchTimeConstraint("TC2");

      if(crc.getTimeConstraints.TimeConstraintAt(j).isFire())
      {
         int i = crc.getPorts().searchPort("null");

         crc.getPorts().portAt(i).Send(crc, new Event("Up", "Internal", "null"));
         crc.getTimeConstraints.TimeConstraintAt(j).disactive();
      }
   }
   }
}
```

Figure 65: Implementation for State *toOpen* of Gate

```
class Gate extends CRC
{
   Gate(String n, PortParas pps, Supervisor sp)
   {
      super(n, sp);

      PortPara pp;
      pp = (PortPara)pps.elementAt(0);
      for (int i = 0; i < pp.getPortNum(); i ++)
         addPort(new Port(pp.getportID(i), pp.getConnectTrom(i), pp.getConnectPort(i)));
      addPort(new Port("null", Name, "null"));

      addState("main", new Gate_opened());
      addState("main", new Gate_toClose());
      addState("main", new Gate_closed());
      addState("main", new Gate_toOpen());
      mainState.initialcState();

      addTransition(new Gate_R1());
      addTransition(new Gate_R2());
      addTransition(new Gate_R3());
      addTransition(new Gate_R4());

      addTimeConstraint(new TimeConstraint("TC1", "R1", new Event("Down", "Internal", "null"), 0, 1,
                                "toClose", new StateNames()));
      addTimeConstraint(new TimeConstraint("TC2", "R3", new Event("Up", "Internal", "null"), 1, 2,
                                "toOpen", new StateNames()));
   }
}
```

Figure 66: Implementation for Gate

2. Transition R2: Function *PortCondition* returns true only *pid* is not the member of Set *inSet*. Function *Enable* returns true. Function *Computation* inserts *pid* into *inSent* and keeps the current state. Figure 68 shows the impelementation of R2.

3. Transition R3: Function *PortCondition* and *Enable* just return true. Function *Computation* changes the current state to *idle*. Figure 69 shows the implementation of R3.

4. Transition R4: Function *PortCondition* returns true only *pid* is the member of Set *inSet*. Function *Enable* returns true only the size of *inSet* is 1. Function *Computation* removes *pid* into *inSent* and changes the current state to *deactivate*. Figure 70 shows the impelementation of R4.

5. Transition R5: Function *PortCondition* returns true only *pid* is not the member of Set *inSet*. Function *Enable* returns true. Function *Computation* inserts *pid* into *inSet* and keeps the current state. Figure 71 shows the impelementation of R5.

6. Transition R6: Function *PortCondition* returns true only *pid* is the member of Set *inSet*. Function *Enable* returns true only the size of *inSet* is bigger than 1. Function *Computation* removes *pid* into *inSent* and keeps the current state. Figure 72 shows the impelementation of R6.

7. Transition R7: Function *PortCondition* and *Enable* just return true. Function *Computation* inserts *pid* into inSet and changes the current state to *monitor*. Figure 73 shows the implementation of R7.

State There are four states in Controller.

1. State *idle*: In this state, the controller just waits for an event *Near* from the environment. So there is no action in this state. Figure 74 shows the implementation of state *idle*.

2. State *activate*: There is only two types of events that can happen in this state. One is the incoming event *Near*. The other is the internal event *Lower*. *Lower* is a time-constraint event. Therefore, the event is fired when the setting time is arrived. Figure 75 shows the implementation of state *activate*.

```
class Controller_R1 extends Transition
{
   Controller_R1()
   {
      super("R1", "activate", "monitor", new Event("Lower", "null", "null"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      return(true);
   }

   public void Computation(CRC crc, Event e)
   {
      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 67: Implementation for R1 of Controller

```
class Controller_R2 extends Transition
{
   Controller_R2()
   {
      super("R2", "activate", "activate", new Event("Near", "null", "null"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      if(!(((Controller)crc).inSet.member(crc.PID)))
         return(true);
      else
         return(false);
   }

   public void Computation(CRC crc, Event e)
   {
      ((Controller)crc).inSet.add(crc.PID);

      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 68: Implementation for R2 of Controller

```
class Controller_R3 extends Transition
{
   Controller_R3()
   {
      super("R3", "deactivate", "idle", new Event("Raise", "null", "null"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      return(true);
   }

   public void Computation(CRC crc, Event e)
   {

      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 69: Implementation for R3 of Controller

```
class Controller_R4 extends Transition
{
   Controller_R4()
   {
      super("R4", "monitor", "deactivate", new Event("Exit", "null", "null"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      if(((Controller)crc).inSet.size() == 1)
         return(true);
      else
         return(false);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      if(((Controller)crc).inSet.member(crc.PID))
         return(true);
      else
         return(false);
   }

   public void Computation(CRC crc, Event e)
   {
      ((Controller)crc).inSet.remove(crc.PID);

      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 70: Implementation for R4 of Controller

```
class Controller_R5 extends Transition
{
   Controller_R5()
   {
      super("R5", "monitor", "monitor", new Event("Near", "null", "null"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      if(!(((Controller)crc).inSet.member(crc.PID)))
         return(true);
      else
         return(false);
   }

   public void Computation(CRC crc, Event e)
   {
      ((Controller)crc).inSet.add(crc.PID);

      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 71: Implementation for R5 of Controller

3. State *monitor*: In this state, the controller just waits for two types of events, *Near* and *Exit*, from the environment. So there is no action in this state. Figure 76 shows the implementation of state *monitor*.

4. State *deactivate*: There is only an outgoing event *Raise* that can happens in this state. This event is a time-constraint event. Therefore, the event is fired when the setting time is arrived. Figure 77 shows the implementation of state *deactivate*.

Reactive Class In Gate, there are one attribute, two port-type, four states, seven transitions and two time-constraints. Figure 78 shows the implementation of Controller.

5.3.3 System Class

The Railroad subsystem that we implemented has 5 trains, 2 controllers, and 2 gates. Figure 79 shows the implementation of this system.

```
class Controller_R6 extends Transition
{
   Controller_R6()
   {
      super("R6", "monitor", "monitor", new Event("Exit", "null", "null"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      if(((Controller)crc).inSet.size() > 1)
         return(true);
      else
         return(false);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      if(((Controller)crc).inSet.member(crc.PID))
         return(true);
      else
         return(false);
   }

   public void Computation(CRC crc, Event e)
   {
      ((Controller)crc).inSet.remove(crc.PID);

      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 72: Implementation for R6 of Controller

```
class Controller_R7 extends Transition
{
   Controller_R7()
   {
      super("R7", "idle", "activate", new Event("Near", "null", "null"));
   }

   public boolean Enable(CRC crc, Event e)
   {
      return(true);
   }

   public boolean PortCondition(CRC crc, Event e)
   {
      return(true);
   }

   public void Computation(CRC crc, Event e)
   {
      ((Controller)crc).inSet.add(crc.PID);

      crc.getState().setcState(sourceSN, destinationSN);
   }
}
```

Figure 73: Implementation for R7 of Controller

```
class Controller_idle extends State
{
   Controller_idle()
   {
      super("idle", true, new States());
   }

   public void run(CRC crc)
   {
   }
}
```

Figure 74: Implementation for State *idle* of Controller

```
class Controller_activate extends State
{
   Controller_activate()
   {
      super("activate", false, new States());
   }

   public void run(CRC crc)
   {
      int j =  crc.getTimeConstraints.searchTimeConstraint("TC1");

      if(crc.getTimeConstraints.TimeConstraintAt(j).isFire())
      {
         for(int i = 0; i < crc.getPorts().size(); i ++)
         {
            if(crc.getPorts().portAt(i).getType().equals("@S"))
               ((Train)crc).portAt(i).Send(crc, new Event("Lower"));
         }

         crc.getTimeConstraints.TimeConstraintAt(j).disactive();
      }
   }
}
```

Figure 75: Implementation for State *activate* of Controller

```
class Controller_monitor extends State
{
   Controller_monitor()
   {
      super("monitor", false, new States());
   }

   public void run(CRC crc)
   {
   }
}
```

Figure 76: Implementation for State *monitor* of Controller

```
class Controller_deactivate extends State
{
   Controller_deactivate()
   {
      super("deactivate", false, new States());
   }

   public void run(CRC crc)
   {
      int j =  crc.getTimeConstraints.searchTimeConstraint("TC2");

      if(crc.getTimeConstraints.TimeConstraintAt(j).isFire())
      {
         for(int i = 0; i < crc.getPorts().size(); i ++)
         {
            if(crc.getPorts().portAt(i).getType().equals("@S"))
               ((Train)crc).portAt(i).Send(crc, new Event("Raise"));
         }

         crc.getTimeConstraints.TimeConstraintAt(j).disactive();
      }
   }
}
```

Figure 77: Implementation for State *deactivate* of Controller

5.4 Automatic Code Generation

Our automatic code generation tool requests users input TROM specification file (*.trom), subsystem specification file (*.scs), and Larch/Java specification file (*.java). The codes are automatically generated from them.

In our case study, the codes generated by the tool are identical to manually generated codes. This is due to the fact that the code generation algorithms completely follow our implementation methodology. Therefore, our tool can be considered as a helpful assistant for implementing real-time reactive systems.

It is probable that in some cases the generated code may not work since the specifications may lack detailed information. For instance, if we do not give the condition when a train should fire a *Near* event and at which port should the event be fired in our case study, the whole system can not work. The detailed information may be implemented manually. However, after the automatic code generation tool is developed, this work can become a requirement to refine the original design. When the generated codes fail to work, the programmers will require the designers to give more detailed information by refining the original design, then generate the codes again and test them. By iteratively refining the design, the final implementation can be fine

```
class Controller extends CRC
{
   public Set inSet;

   Controller(String n, PortParas pps, Supervisor sp)
   {
      super(n, sp);

      inSet = new Set();

      int i, j;
      PortPara pp;

      for(j = 0; j < pps.size(); j ++)
      {
         pp = (PortPara)pps.elementAt(j);

         if(pp.getPortType().equals("@P"))
            for (i = 0; i < pp.getPortNum(); i ++)
               addPort(new Port(pp.getportID(i), pp.getConnectTrom(i), pp.getConnectPort(i)));
         if(pp.getPortType().equals("@G"))
            for (i = 0; i < pp.getPortNum(); i ++)
               addPort(new Port(pp.getportID(i), pp.getConnectTrom(i), pp.getConnectPort(i)));
      }
      addPort(new Port("null", Name, "null"));

      addState("main", new Controller_idle());
      addState("main", new Controller_activate());
      addState("main", new Controller_deactivate());
      addState("main", new Controller_monitor());
      mainState.initialcState();

      addTransition(new Controller_R1());
      addTransition(new Controller_R2());
      addTransition(new Controller_R3());
      addTransition(new Controller_R4());
      addTransition(new Controller_R5());
      addTransition(new Controller_R6());
      addTransition(new Controller_R7());

      addTimeConstraint(new Controller_TCvar1());
      addTimeConstraint(new Controller_TCvar2());
   }

   public void display_attributes(PrintStream out)
   {
      out.println(" There is 1 attribute in this TROM object.");
      out.println("    1th Attribute Name : inSet");
      out.println("        Attribute Value: " + inSet);
   }
}
```

Figure 78: Implementation for Controller

```
class TCG
{
   public static void main(String arg[])
   {
      RTSuper rtsuper = new RTSuper();

      PortParas pps;
      PortPara pp;

      pps = new PortParas();
      pp.setPortType("GC");
      pp.setPortNum(1);
      pp.setConnection(0, "C1", "Controller1", "P1");
      Train train1= new Train("train1", pps, rtsuper);

      pps = new PortParas();
      pp.setPortType("GC");
      pp.setPortNum(1);
      pp.setConnection(0, "C2", "Controller1", "P2");
      Train train2= new Train("train2", pps, rtsuper);

      pps = new PortParas();
      pp.setPortType("GC");
      pp.setPortNum(2);
      pp.setConnection(0, "C3", "Controller1", "P3");
      pp.setConnection(1, "C4", "Controller2", "P4");
      Train train3= new Train("train3", pps, rtsuper);

      pps = new PortParas();
      pp.setPortType("GC");
      pp.setPortNum(1);
      pp.setConnection(0, "C5", "Controller2", "P5");
      Train train4= new Train("train4", pps, rtsuper);

      pps = new PortParas();
      pp.setPortType("GC");
      pp.setPortNum(1);
      pp.setConnection(0, "C6", "Controller2", "P6");
      Train train5= new Train("train5", pps, rtsuper);

      pps = new PortParas();
      pp.setPortType("GP");
      pp.setPortNum(3);
      pp.setConnection(0, "P2", "train2", "C2");
      pp.setConnection(1, "P1", "train1", "C1");
      pp.setConnection(2, "P3", "train3", "C3");
      pp.setPortType("GG");
      pp.setPortNum(1);
      pp.setConnection(0, "G1", "Gate1", "S1");
      Controller Controller1= new Controller("Controller1", pps, rtsuper);

      pps = new PortParas();
      pp.setPortType("GP");
      pp.setPortNum(3);
      pp.setConnection(0, "P5", "train4", "C5");
      pp.setConnection(1, "P6", "train5", "C6");
      pp.setConnection(2, "P4", "train3", "C4");
      pp.setPortType("GG");
      pp.setPortNum(1);
      pp.setConnection(0, "G2", "Gate2", "S2");
      Controller Controller2= new Controller("Controller2", pps, rtsuper);

      pps = new PortParas();
      pp.setPortType("GS");
      pp.setPortNum(1);
      pp.setConnection(0, "S1", "Controller1", "G1");
      Gate Gate1= new Gate("Gate1", pps, rtsuper);

      pps = new PortParas();
      pp.setPortType("GS");
      pp.setPortNum(1);
      pp.setConnection(0, "S2", "Controller2", "G2");
      Gate Gate2= new Gate("Gate2", pps, rtsuper);

      rtsuper.start();
   }
}
```

Figure 79: Implementation for Train-Gate-Controller System

tuned. In most of cases, lack of detailed information is associated with unconstraint events. The designers have to find out these events and refine them to give more precise information on where and when they can occur. The main contribution of this tool is to reduce the implementation time and increase the quality of code. The whole implementation phase becomes as shown in Figure 80.

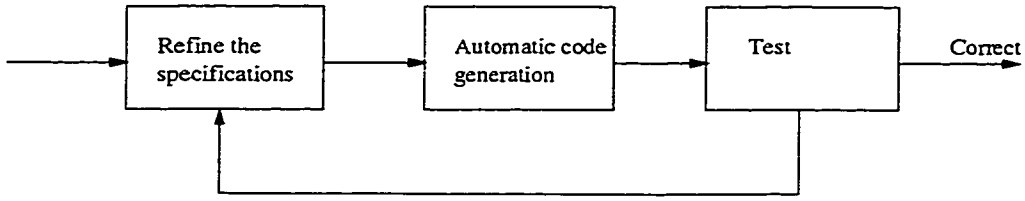


Figure 80: Implementation Phase

The design notation scales up easily, that is, a large system can be specified succinctly in TROM notation. In Figure 81, *Train20* crosses *Gate1*, *Gate2*, *Gate3*, *Gate4*, and *Gate5*. An implementation of the object *Train20* will inherit from the code *Train* class and refine the data member sensor to provide the addition information on the sequence of gates that the train has to cross. As an example, give *Sensor* is refined to the *Vector* <2, 3, 5, 4, 1>, then *Train20* will cross *Gate2*, *Gate3*, *Gate5*, *Gate4*, *Gate1* in that order. However, if *Train20* does not cross all the gates, it is sufficient to give the order in which it is crossing the other gates. As an example if in the subsystem *Train20* crosses *Gate2* and *Gate3* in that order, and crosses no other gate, then *Sensor* is refined to *Vector* <2,3>. To sum up for correct implementation of the system from system specification, it is necessary to provide the additional information on the exact ordering in which train will cross the gate.

```
SCS TrainGateController2
    Includes:
    Instantiate:
        Gate1::Gate[@S:1];
        Gate2::Gate[@S:1];
        Gate3::Gate[@S:1];
        Gate4::Gate[@S:1];
        Gate5::Gate[@S:1];
        Controller1::Controller[@P:7, @G:1];
        Controller2::Controller[@P:8, @G:1];
        Controller3::Controller[@P:8, @G:1];
        Controller4::Controller[@P:7, @G:1];
        Controller5::Controller[@P:6, @G:1];
        train1::Train[@C:1];
        train2::Train[@C:1];
        train3::Train[@C:1];
        train4::Train[@C:3];
        train5::Train[@C:2];
        train6::Train[@C:1];
        train7::Train[@C:1];
        train8::Train[@C:4];
        train9::Train[@C:2];
        train10::Train[@C:1];
        train11::Train[@C:1];
        train12::Train[@C:2];
        train13::Train[@C:2];
        train14::Train[@C:1];
        train15::Train[@C:1];
        train16::Train[@C:2];
        train17::Train[@C:3];
        train18::Train[@C:1];
        train19::Train[@C:1];
        train20::Train[@C:5];
    Configure:
        Gate1.@S1:@S ↔ Controller1.@G1:@G;
        Gate2.@S2:@S ↔ Controller2.@G2:@G;
        Gate3.@S3:@S ↔ Controller3.@G3:@G;
        Gate4.@S4:@S ↔ Controller4.@G4:@G;
        Gate5.@S5:@S ↔ Controller5.@G5:@G;
        train1.@C1:@C ↔ Controller1.@P1:@P;
        train2.@C2:@C ↔ Controller1.@P2:@P;
        train3.@C3:@C ↔ Controller1.@P3:@P;
        train4.@C4:@C ↔ Controller1.@P4:@P;
        train4.@C5:@C ↔ Controller2.@P8:@P;
        train4.@C6:@C ↔ Controller3.@P16:@P;
        train5.@C7:@C ↔ Controller1.@P5:@P;
        train5.@C8:@C ↔ Controller2.@P9:@P;
        train6.@C9:@C ↔ Controller2.@P10:@P;
        train7.@C10:@C ↔ Controller2.@P11:@P;
        train8.@C11:@C ↔ Controller2.@P12:@P;
        train8.@C12:@C ↔ Controller3.@P17:@P;
        train8.@C13:@C ↔ Controller4.@P24:@P;
        train8.@C14:@C ↔ Controller5.@P31:@P;
        train9.@C15:@C ↔ Controller2.@P13:@P;
        train9.@C16:@C ↔ Controller3.@P18:@P;
        train10.@C17:@C ↔ Controller3.@P19:@P;
        train11.@C18:@C ↔ Controller3.@P20:@P;
        train12.@C19:@C ↔ Controller3.@P21:@P;
        train12.@C20:@C ↔ Controller4.@P25:@P;
        train13.@C21:@C ↔ Controller2.@P14:@P;
        train13.@C22:@C ↔ Controller4.@P26:@P;
        train14.@C23:@C ↔ Controller4.@P27:@P;
        train15.@C24:@C ↔ Controller4.@P28:@P;
        train16.@C25:@C ↔ Controller4.@P29:@P;
        train16.@C26:@C ↔ Controller5.@P32:@P;
        train17.@C27:@C ↔ Controller1.@P6:@P;
        train17.@C28:@C ↔ Controller3.@P22:@P;
        train17.@C29:@C ↔ Controller5.@P33:@P;
        train18.@C30:@C ↔ Controller5.@P34:@P;
        train19.@C31:@C ↔ Controller5.@P35:@P;
        train20.@C32:@C ↔ Controller1.@P7:@P;
        train20.@C33:@C ↔ Controller2.@P15:@P;
        train20.@C34:@C ↔ Controller3.@P23:@P;
        train20.@C35:@C ↔ Controller4.@P30:@P;
        train20.@C36:@C ↔ Controller5.@P36:@P;
end
```

Figure 81: Formal specification for subsystem TrainGateController2

Chapter 6

Conclusions and Future Work

Real-time reactive systems are time-critical. Hence 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. It is also impossible to make available all necessary computing resources in order to guarantee the appropriate output under all possible conditions. In large industrial applications real-time systems are usually constructed using processors with considerable spare capacity, thereby ensuring that "worst-case behavior" does not produce any unwelcome delays during critical periods of the system's operation. In this thesis, such limiting factors have not been considered. We have focused on two major implementation issues:

- using Real-time Java as a high-level implementation language
- automating the code generation in Real-time Java from TROM specifications

The correctness of the implementation with respect to the TROM specifications are informally justified through the mapping between the models in the specification tier and the models in the implementation tier. The logic behind the algorithms that generate code from the specifications closely follow the operational and logical semantics for TROM - based systems [Ach95]. A formal proof of correctness of the implementation may be too hard to work out, and is definitely outside the scope of

this thesis. However, the implemented code can be tested against the specification by the testing method that has recently been developed [AOM00].

The language Java was a natural choice of implementation for the following reasons:

OO The design notation is object-oriented and Java is a pure OO language. This makes the mapping of the design constructs to the language constructs, in an automatic code generation exercise, easy to comprehend, maintain, and verify. The verification process can be only informal, yet it will be convincing due to the seamless meshing of the conceptual boundaries.

Security The run-time support system and Java compiler can automatically detect programming errors.

Portability The program is independent of the hardware on which it runs; however the hardware resources that the program should manipulate (such as environmental objects) are closely tied to the program behavior.

Testability The modularity that is present in the design is preserved in the implementation model. Consequently, it is easy to identify the component in the implementation that can be tested with test data associated with the test templates generated from a design specification component. In addition to unit testing, concurrent and communicating sequential activities can be tested with the test templates generated from the subsystem configuration specification for which the unit under test is the implementation. A simulator [Hai99] to simulate and reason about system activities has been implemented in Java. The simulator and our implementation can be run in parallel, and abnormal as well as "normal" system behavior created by the simulator can be automatically fed to the implementation to test the corresponding behavior in the implemented system.

Evolution The Java implementation can be easily enhanced as the design evolves due to changing requirements and design refinements.

Language design is still an active research area. Although the design should naturally lead to an implementation, often the expressive power of modern programming languages do not match current design methodologies. Some of the concurrent programming languages used for implementing real-time systems are Ada, Modula-2,

Occam 2, and C. The use of Java in this thesis should be noted as one of the first attempts to implement real-time reactive systems. Moreover, an automatic code generator from a formal specification into Java has not been reported in literature. So, on both counts the work presented in this thesis is quite new.

Some of the future extensions to our work include the following:

1. Integrating the Java code generator with the simulator and test generator programs within the TROMLAB environment;

2. Enhancing the code generation corresponding to transition specifications to handle parameterized events in the specification;

3. Empirically evaluating the Java code generator for large size problems, where both the number of objects and the number of interactions are large.

4. Incorporate forward and backward error recovery schemes and provide fault treatment.

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.

Appendix A

GRC and SCS Grammar

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

Table 2: Grammar for generic reactive class specification

In the grammar, a class (see Table 3) is described by the keyword Class, followed by a string denoting the class name, followed by a list of port types in square brackets . The list of port types is composed of one or several port type names, represented as strings starting with the symbol @ and separated by a comma.

class	::=	Class <class_name> [<port_types>] NL
port_types	::=	<port_type_name> | <port_type_name>, <port_types>
class_name	::=	String
port_type_name	::=	@String

Table 3: Grammar for generic reactive class title

Events (see Table 4) are introduced by the keyword Events, followed by the list of events. The list of events can contain one or several events, separated by comma. Each event can be an internal event, an input event or an output event. Internal events are represented by a string for the event name. Input events are represented by a string as event name, followed by the character ? and the string for the port type at which the event occurs. Output events are represented by a string as event name, followed by the character ! and the string for the port type at which the event occurs.

States (see Table 5) are introduced by the keyword States, followed by the state set. The state set is comprised of the initial state, followed by a list of one or several states, separated by comma. A state is represented by a string for the name. If the state is complex, the name is followed by its substates, represented as a state set, within curly braces.

events	::=	Events: <event_list> NL
event_list	::=	<event> | <event>, <event_list>
event	::=	<inputevent> | <outputevent> | <interevent>
inputevent	::=	<event_name> ? <port_type_name>
outputevent	::=	<event_name> ! <port_type_name>
interevent	::=	<event_name>
event_name	::=	String
port_type_name	::=	@String

Table 4: Grammar for events

states	::=	States: <state_set> NL
state_set	::=	*<state>, <state_list>
state_list	::=	<state> | <state>, <state_list>
state	::=	<state_name> | <state_name><state_set>
state_name	::=	String

Table 5: Grammar for states

Attributes (see Table 6) are introduced by the keyword Attributes, followed by the list of attributes. The list of attributes is comprised of one or several attributes, separated by a semi-colon. Attributes of type port type are represented by a string for the attribute name, followed by colon and by the port type name, which starts with the character @. Attributes of type data type are represented by a string for the attribute name, followed by a colon and by the LSL trait type name.

LSL traits (see Table 7) are introduced by the keyword Traits, followed by a list of traits. The list of traits is comprised of one or several traits. A trait is represented as a string for the trait name, followed in square brackets by the argument list and

attributes	::=	Attributes: <att_list>NL
att_list	::=	<attribute> | <attribute>;<att_list>
attribute	::=	<att_name> : <port_type_name> | <att_name> : <trait_type_name> | <att_name> : Integer | <att_name> : Boolean
att_name	::=	String
trait_type_name	::=	String
port_type_name	::=	@String

Table 6: Grammar for attributes

traits	::=	Traits: <trait_list> NL
trait_list	::=	<trait> | <trait>, <trait_list>
trait	::=	<trait_name>[<arg_list>,<trait_type_name>] | <trait_name>[<trait_type_name>]
arg_list	::=	<arg> | <arg>, <arg_list>
arg	::=	<trait_type_name> | <port_type_name>
trait_name	::=	String
trait_type_name	::=	String
port_type_name	::=	@String

Table 7: Grammar for LSL traits

att_funcs	::=	Attribute–Function: <att_func_list>
att_func_list	::=	<att_func>; | <att_func>;<att_func_list>
att_func	::=	<state_name> → <att_list> NL
att_list	::=	<att_name> | <att_name>,<att_list> | empty
att_name	::=	String
state_name	::=	String

Table 8: Grammar for attribute functions

the trait type name. The argument list is comprised of one or several arguments. An argument is either a trait type name or a port type name starting with the character @.

The attribute function (see Table 8) is introduced by the keyword Attribute-Function, followed by a list of attribute function applications. The list of attribute function applications has one or several attribute function applications, separated by a semi-colon. Each attribute function application is comprised of the state name as a string, followed by the keyword →, followed by an attribute list, between curly braces. An attribute list is comprised of zero or several attribute names, separated by a comma.

Transition specifications (see Table 9) are introduced by the keyword Transition-Specifications, followed by the list of transition specifications, separated by semi-colons and new lines. The list of transition specifications is composed of one or several transition specifications, separated by new lines. A transition specification consists of a name, followed by a colon, one or several state pairs, separated by semi-colons, a triggering event, an assertion, the implication operator → and another assertion. A state pair consists of two state names, in brackets, separated by a comma. The triggering event is an event name followed in brackets by an assertion. An assertion is either a

tran_specs	::=	Transition–Specifications: NL <tran_spec_list>
tran_spec_list	::=	<tran_spec> NL | <tran_spec> NL <tran_spec_list>
tran_spec	::=	<tran_spec_name>: <state_pairs> <trig_event> <assertion> → <assertion>;
state_pairs	::=	<state_pair>; | <state_pair>; <state_pairs>;
state_pair	::=	(<state_name>,<state_name>)
trig_event	::=	<event_name>(<assertion>)
assertion	::=	<simple_exp> | <simple_exp> <b_op> <simple_exp>
b_op	::=	= | ≠ | > | ≥ | < | ≤
simple_exp	::=	<term> | <term> <OR> <term>
term	::=	<factor> | <factor> <AND> <factor>
factor	::=	<NOT> <factor> | pid | <att_name/ > | <att_name> | true | false | <LSL_term> | (<assertion>)
LSL_term	::=	<LSL_func_name>(<arg_list>)
arg_list	::=	<arg>|<arg>,<arg_list>
arg	::=	pid | <att_name> | <LSL_term>
att_name/	::=	String
att_name	::=	String
state_name	::=	String
event_name	::=	String
LSL_func_name	::=	String
OR	::=	|
AND	::=	&
NOT	::=	!

Table 9: Grammar for transition specifications

simple expression or two simple expressions with a binary operator between them. A binary operator is one of: $=$, $\neq$, $<$, $\leq$, $>$, $\geq$. A simple expression is either a term or two terms with the | logical operator. A term is either a factor, or two factors with the & logical operator. A factor can be the logical operator ! followed by a factor, or the reserved variable pid, or a primed attribute, an attribute, logical expressions *true* or *false*, an LSL term or an assertion in brackets. An LSL term consists of a LSL function name, followed by an argument list in brackets. An argument list is composed of one or several arguments. An argument is either the reserved variable pid, or an attribute name or an LSL term. A primed attribute is an attribute (from the attribute function) followed by the character /.

Time constraints (see Table 10) are introduced by the keyword Time-Constraints, followed by one or several constraints, separated by semi-colons and new lines. A

time_constraints	::=	Time−Constraints: NL <constraints>
constraints	::=	<constraint>; NL | <constraint> ; NL <constraints>
constraint	::=	<time_cons_name>: <tran_spec_name>, <event_name>, <min_type><min>,<max><max_type>,<states>
states	::=	<state_name>|<state_name>,<states> | empty
state_name	::=	String
time_cons_name	::=	String
tran_spec_name	::=	String
event_name	::=	String
min	::=	NAT
max	::=	NAT
min_type	::=	(| [
max_type	::=	) |]

Table 10: Grammar for time constraints

SCS	::=	SCS <scs_name> NL <include> <instantiates> <configure> end
scs_name	::=	String

Table 11: Grammar for subsystem configuration

constraint has a name followed by colon and the name of the constraining transition specification, the name of the constrained event, the lower and upper bounds, and a list of disabling states. The lower and upper bounds are preceded and followed, respectively, by the open or closed interval indicators. The list of disabling states is comprised of zero, one or several state names, separated by a comma.

The configuration specification should respect the following grammar, introduced in [Tao96].

A subsystem configuration specification (see Table 11) is introduced by the keyword SCS, followed by its name as a string, a new line and the following sections: Includes, Instantiates, Configure, all followed by the keyword end.

The include section (see Table 12) is introduced by the keyword Includes, followed by a list of subsystem names and a new line. The list of subsystem names is composed of one or several subsystem names, separated by a semi-colon.

The instantiates section (see Table 13) is introduced by the keyword Instantiate, followed by an instance list and a new line. An instance list is composed of one or several

include	::=	Includes: <scs_name_list> NL
scs_name_list	::=	<scs_name>; | <scs_name_list>
scs_name	::=	String

Table 12: Grammar for include section

instantiates	::=	Instantiate: <inst_list> NL
inst_list	::=	<instantiate>; NL | <instantiate>; NL <inst_list>
instantiate	::=	<obj_name>::<grc_name>[<port_card_list>]
port_card_list	::=	<port_card>|<port_card>,<port_card_list>
port_card	::=	<port_type_name>:<cardinality>
obj_name	::=	String
port_type_name	::=	@String
grc_name	::=	String
cardinality	::=	NAT

Table 13: Grammar for instantiate section

instances. An instance consists of an object name, followed by two colons, a generic class name and, in square brackets, by a port cardinality list. The port cardinality list is composed of one or several port cardinalities. A port cardinality is represented by a port type name, followed by a colon and a natural number for the cardinality.

The configure section (see Table 14) is introduced by the keyword Configure, followed by the object port list. The object port list is composed by one or several object port links, separated by a semi-colon. An object port link is composed of an object name, followed by a period, a port name starting with character @ and its port type, the composition operator $\leftrightarrow$, another object name, followed by a period, and a port name starting with character @ and its port type.

configure	::=	Configure: <obj_port_list>
obj_port_list	::=	<obj_port_link>; NL | <obj_port_link>; NL <obj_port_list>;
obj_port_link	::=	<obj_name>.<port_name>:<port_type_name> $\leftrightarrow$ <obj_name>.<port_name>:<port_type_name>
obj_name	::=	String
port_name	::=	@String
port_type_name	::=	@String

Table 14: Grammar for configure section

Appendix B

Grammer of Larch/Java

The syntax of Larch/Java is combination of two styles: the first style has Java notation for declaring names and functions; the second style has Larch/ C++ notation for describing the behavior of the functions. The syntax accepted by the LIL parser in this thesis is shown below:

```
Larch_Java_Specification ::= CompilationUnit

CompilationUnit ::= [PackageDeclaration]
                    (ImportDeclaration)*
                    (TypeDeclaration)*

PackageDeclaration ::= ''package'' Name '';''

ImportDeclaration ::= ''import'' Name [''.'' ''*''] '';''

TypeDeclaration ::= ClassDeclaration |
                    InterfaceDeclaration |
                    '';''

ClassDeclaration ::= (''abstract'' | ''final'' | ''public'')*
                     UnmodifiedClassDeclaration

UnmodifiedClassDeclaration ::= ''class'' IDENTIFIER
                               [''extends'' Name]
                               [''implements'' NameList]
                               ClassBody

ClassBody ::= ''{'' (ClassBodyDeclaration)* ''}''

NestedClassDeclaration ::= (''static'' | ''abstract'' | ''final'' |
                            ''public'' | ''protected'' | ''private'')*
                            UnmodifiedClassDeclaration

ClassBodyDeclaration ::= Initializer |
                         LarchUsesDeclaration |
                         NestedClassDeclaration |
                         NestedInterfaceDeclaration |
                         ConstructorDeclaration |
                         MethodDeclaration


LarchUsersDeclaration ::= ''uses'' LarchTraitList() '';''
```

```
LarchTraitList ::= IDENTIFIER [''('' LarchRenaming '')''']

LarchRenaming ::= IDENTIFIER ''for'' IDENTIFIER (''','' IDENTIFIER ''for'' IDENTIFIER)*

InterfaceDeclaration ::= (''abstract'' | ''public'')*
                         UnmodifiedInterfaceDeclaration

NestedInterfaceDeclaration ::= (''static'' | ''abstract'' | ''final'' |
                                ''public'' | ''protected'' | ''private'')*
                                UnmodifiedInterfaceDeclaration

UnmodifiedInterfaceDeclaration ::= ''interface'' IDENTIFIER
                                    [''extends'' NameList]
                                    ''{'' (InterfaceMemberDeclaration)* ''}''

InterfaceMemberDeclaration ::= NestedClassDeclaration |
                                NestedInterfaceDeclaration |
                                MethodDeclaration

MethodDeclaration ::= (''public'' | ''protected'' | ''private'' | ''static'' | ''abstract''
                       | ''final'' | ''native'' | ''synchronized'')*
                       ResultType
                       MethodDeclarator [''throws'' NameList]
                       (Block | '';'')

MethodDeclarator ::= IDENTIFIER FormalParameters (''['' '']'')*

FormalParameters ::= ''('' [FormalParameter (''','' FormalParameter)*] '')''

FormalParameter ::= [''final''] Type VariableDeclaratorID

ConstructorDeclaration ::= [''public'' | ''protected'' | ''private'']
                           IDENTIFIER FormalParameters [''throws'' NameList]
                           ''{''
                           (BlockStatement)*
                           ''}''

Initializer ::= [''static''] Block

Type ::= PrimitiveType (''['' '']'')*

PrimitiveType ::= ''boolean'' | ''char'' | ''byte'' | ''short'' |
                  ''int'' | ''long'' | ''float'' | ''double''

ResultType ::= ''void'' | Type

Name ::= IDENTIFIER (''.'' IDENTIFIER)*

NameList ::= Name (''','' Name)*

Block ::= ''{'' (BlockStatement)* ''}''
```

```
BlockStatement ::= Statement |
                   UnmodifiedClassDeclaration |
                   UnmodifiedInterfaceDeclaration

Statement ::= LarchRequiresClause |
              LarchModifiesClause |
              LarchEnsuresClause

LarchRequiresClause ::= ``requires'' [LarchPredicate] ``;''

LarchModifiesClause ::= ``modifies'' [IDENTIFIER (``,'' IDENTIFIER)*] ``;''

LarchEnsuresClause ::= ``ensures'' [LarchPredicate] ``;''

LarchPredicate ::= LarchLogicalTerm

LarchLogicalTerm ::= LarchEqualityTerm (LarchLogicalOpr LarchEqualityTerm)*

LarchEqualityTerm ::= LSLEquality | LSLFunction

LSLEquality ::= IDENTIFIER [LarchStateFunc] LarchEqOpr IDENTIFIER ``('' [LarchParameters] ``)''

LSLFunction ::= IDENTIFIER ``('' [LarchParameters] ``)''

LSLParameters ::= IDENTIFIER [LarchStateFunc] (``,'' IDENTIFIER [LarchStateFunc])*

LarchLogicalOpr ::= ``\and'' | ``\or'' | ``\implies'' | ``/\'' | ``\/'' | ``=>''

LarchEqOpr ::= ``='' | ``==`` | ``\eq'' | ``~='' | ``!='' | ``\neq''

LarchStateFunc ::= ``~'' | ``\pre'' | ``\post''
```