INFORMATION TO USERS

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

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

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

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

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

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

PARAMETERIZED EVENTS FOR DESIGNING REAL-TIME REACTIVE SYSTEMS

May Haydar

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

February 2001

Abstract

Parameterized Events for Designing Real-Time Reactive Systems

May Haydar

The goal of the thesis is the introduction of parameterized events in TROM methodology, an object-oriented formalism for the development of real-time reactive systems. Without parameterization, a large number of events representing similar stimuli or responses are required for modeling large systems, especially in telephony and protocol application domains. The syntax and semantics for parameterized events are conservative extensions of the syntax and semantics of events in TROM formalism. The thesis addresses the impact of parameterization on existing tools of TROMLAB, a framework for practicing TROM formalism, and discusses the reengineering of some of the tools in TROMLAB. The expressive power of the extended formalism is illustrated using the ATM-ABR protocol. The ATM model is validated in SDL. by mapping its TROM model into SDL and executing the SDL model.

To my husband Hesham and my daughter Riham.

Acknowledgments

I would like to thank my supervisor, Dr. V. S. Alagar, for the help, and technical and financial support he had provided through my thesis work. He was always available for any discussion or consultation where he guided this work with patience and understanding.

I would like to thank also Dr. F. Khendek who helped me finish an important task of this thesis by providing me access to his research lab and resources namely Object-GEODE tools and references.

Also, I would like to thank Dr. J. Atwood for his insights and advice at the early stages of the thesis.

A warm thank you goes to my husband for his continuous support and help. Without his encouragement, understanding, and thoughtful love this work would have never been accomplished. To my daughter, a special thank you and I hope I can compensate her for the times I did not spend with her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Real Time Reactive Systems

Reactive systems, as the name implies, interact continuously with their environment. This interaction is based on a stimulus-response behavior that requests a response by the reactive system to each excitation from its environment. When its responses are bounded by strict timing constraints, the system is called a real-time reactive system. The stimulus-response behavior implies that a process in a reactive system can run continuously and infinitely, hence the infinite behavior that characterizes such systems. In this respect, real-time reactive systems are different from interactive systems, although they share some aspects in the sense that the environment and processes in interactive systems have synchronization abilities. However, in a reactive system, such as a shutdown system in a nuclear power plant, a process is totally and solely responsible for the synchronization with its environment. That is, in a reactive system, a process is fast enough to react to stimulus from the environment, and the time between stimulus and response is acceptable enough for the dynamics of the environment to be receptive to the response. Typical applications of real-time reactive systems cover several fields such as telephony, air traffic control, nuclear power reactors, and avionics. In general, real-time reactive systems are required to interact with safety-critical environments where failure leads to major losses. Therefore, such systems must undergo strict and thorough analysis of both the functional and timing properties in order to satisfy the safety requirements. Such analysis usually requires

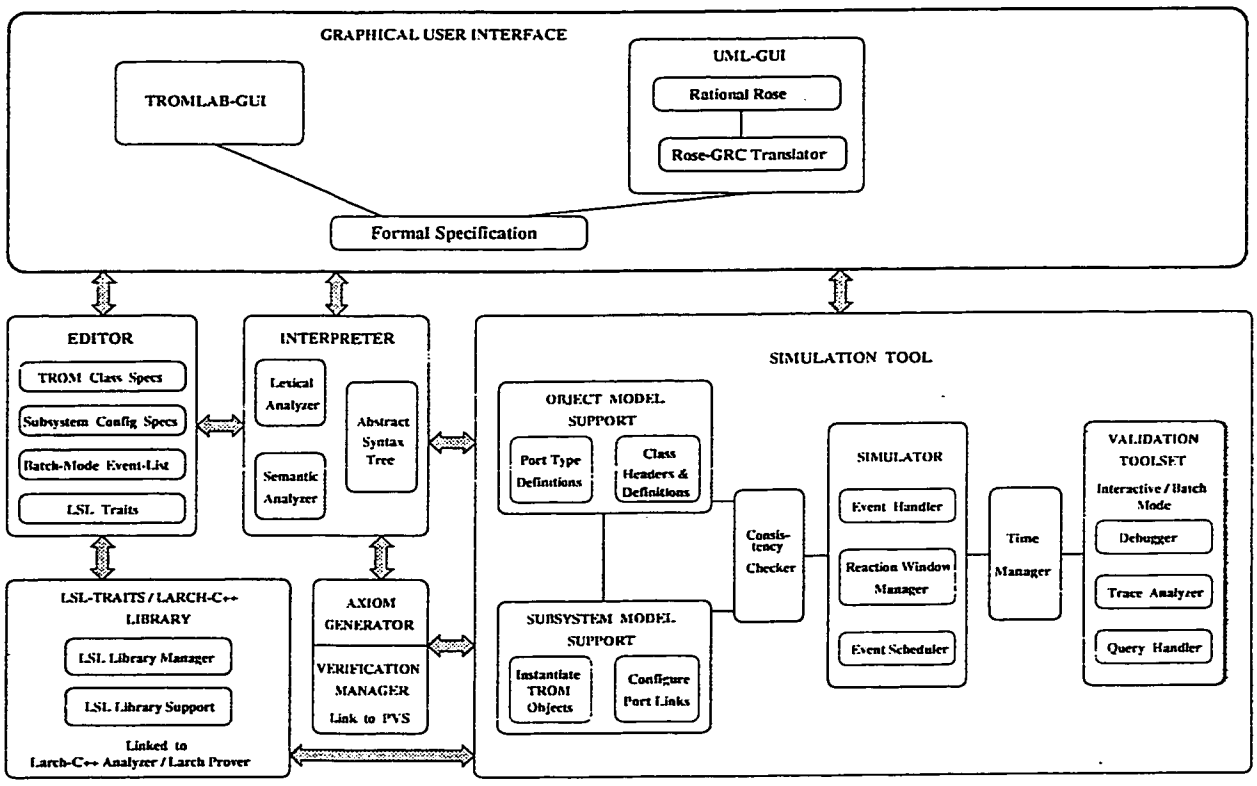

Figure 1: TROMLAB Architecture

more clarity and expressiveness of the models used to represent these systems, especially when they become complex and large.

The behaviour of the reactive system is a combination of its functional behaviour, causal dependencies of actions, and real-time duration governing them. Due to these three layers of interaction, understanding and reasoning about the behaviour of real-time reactive systems becomes difficult.

Timed Reactive Object Model (TROM) is a formalism introduced in [Ach95]. It describes functional and timing properties of real-time reactive systems using formal notations. Formal notations are characterized by their understandability and adequacy to resolve complexity in modeling and designing. A complete semantics of the formalism, and several case studies for illustrations on the formalism have appeared in [Ach95], and [AAR95]. TROMLAB is a development environment based on TROM formalism. Figure 1 is an overall architectural view of TROMLAB.

Currently, TROMLAB includes the following components:

- **Interpreter** - [Tao96] It parses and syntactically checks a TROM specification, and constructs an internal representation;
- **Simulator** - [Mut96] It simulates a subsystem based on the internal representation and enables a systematic validation of the specific system;
- **Browser** - [Nag99] It helps the users navigate, query, and access various system components for reuse during system development;
- **UML-RT Support** - [Pop99] It is a translator that generates TROM specifications from Real-time UML;
- **Verification Assistant** - [Pom99] It generates a PVS theory from TROM specification for proving timing properties;
- **Graphical User Interface** - [Sri99] It is a visual modeling and interaction facility to use the TROMLAB environment.
- **Reasoning System** - [Hai99] It provides a means to debug the system during simulation by allowing interactive queries of a hypothetical nature on system behavior;
- **Code Generator** - [Zha00] It generates Java code from TROM specifications.

1.2 Motivation and Context

Although the use of a formal model helps reduce the complexity in representing and designing real-time reactive systems, the TROM formalism as all state based models may suffer from the problem of state and event explosion when the represented system is relatively large and complex. In particular, the resource explosion problem becomes heavily in focus when it comes to analyzing and verifying the obtained models, e.g., performing reachability analysis on the state machine. In this thesis, we are concerned with solving the state explosion problem in TROM, adding more clarity to the system model, and enhancing the expressiveness of the formalism. We propose to introduce parameters into the events of TROM.

1.3 Scope of the Thesis

This thesis work has been concentrated on defining appropriate syntax and semantics of parameterized events within the TROM formalism. This necessitates redefining the grammar of the TROM formal representation. These changes require modifying the design and implementation of some TROMLAB tools to accommodate the representation and behavior of the parameters. Throughout the thesis, we remodel the Train-Gate-Controller system, a benchmark example studied by real-time reactive systems community, to illustrate how the modifications are carried out. The main contributions of this thesis are:

1. Defining clear syntax and semantics for parameterized events within the TROM formalism.

2. Extending the grammar of TROM formal representation to include the new grammar of the parameters formal representation.

3. Extending the Real-time UML notation to model parameterized events.

4. Reengineering the Rose-GRC Translator to automatically generate the formal representation of parameters.

5. Reengineering the Interpreter to parse, syntactically check, and construct the internal representation of parameters.

As a case study, we remodel the Asynchronous Transfer Mode Available Bit Rate (ATM-ABR) protocol, a Traffic Management service to control and avoid congestion in an ATM network, based on a previous attempt to produce formal specifications of the protocol [LDU96]. The original description of this protocol is provided by the ATM forum [ATM96] as an English specification. We consider a network system composed of a source, a destination, and a switch, and we generate a formal specification of the system based on the English specification.

The presentation of our work in this thesis proceeds as follows: Chapter 2 presents an overview of the TROM formalism along with the components of TROMLAB. In Chapter 3, we show how to extend the formal syntax and semantics of TROM to accommodate parameters. Chapter 4 presents the extensions to the Real-Time UML

notation and the grammar of TROM formal representation. In addition, it shows how to perform the reengineering of both the GRC-Translator and the Interpreter. The chapter also includes an example, the Train-Gate-Controller system, to illustrate how to introduce parameters in a reactive system model. Chapter 5 presents a case study on modeling the ATM-ABR protocol in TROM. In Chapter 6, we present a mapping of the ABR model into SDL (Specification and Description Language) where we verify its correctness using the ObjectGEODE tool set. Finally, the conclusions of our work and the possible future steps are presented in Chapter 7.

Chapter 2

TROMLAB Environment - a Brief Review of the Initial Design

TROMLAB is a formal framework for rigorous modeling and analysis of real-time reactive systems. The process model in TROMLAB framework supports an iterative cycle of formal design, validation, and verification. The iterative development approach of the process model has several benefits:

- it reduces risks by exposing them early in the development process;
- it gives importance to the architecture of the software unit; and
- it allows designing modules for large scale software reuse.

The design of TROMLAB itself is based on object-oriented principles, allowing modular design of TROM classes and modular composition of objects. This enables powerful analysis capabilities to be applied to simpler components, and by compositional principle ensure the correctness of the system.

In this chapter an overview of TROMLAB features and its underlying TROM formalism are give.

2.1 Three-Tiered Design Formalism

The basic building block of a reactive system is TROM, a Timed Reactive Object Model. A TROM is a generic class from which several reactive objects can be instantiated, and combined to create a reactive subsystem. A TROM class may include

abstract data types. Based on this simple scheme, Achuthan [Ach95] proposed a three tier structure for designing real-time reactive systems. This three-tiered design scheme shown in Figure 2 has a language of specification for each tier and a method for combining the design units in different tiers. TROM is a hierarchical finite state machine augmented with ports, attributes, logical assertions on the attributes, and time constraints. The middle-tier formalism specifies TROM classes. Abstract data types are specified as LSL(Larch Shared Language) traits in the lowest tier, and can be used by objects modeled by TROM. The upper-most tier specifies object collaboration where each object is an instance of a TROM. Lower-tier specifications are imported into higher-level specifications. The benefits gained by using object-oriented techniques include modularity, reuse, encapsulation, and hierarchical decomposition based on inheritance.

The three tiers shown in Figure 2 are briefly described in the following three subsections.

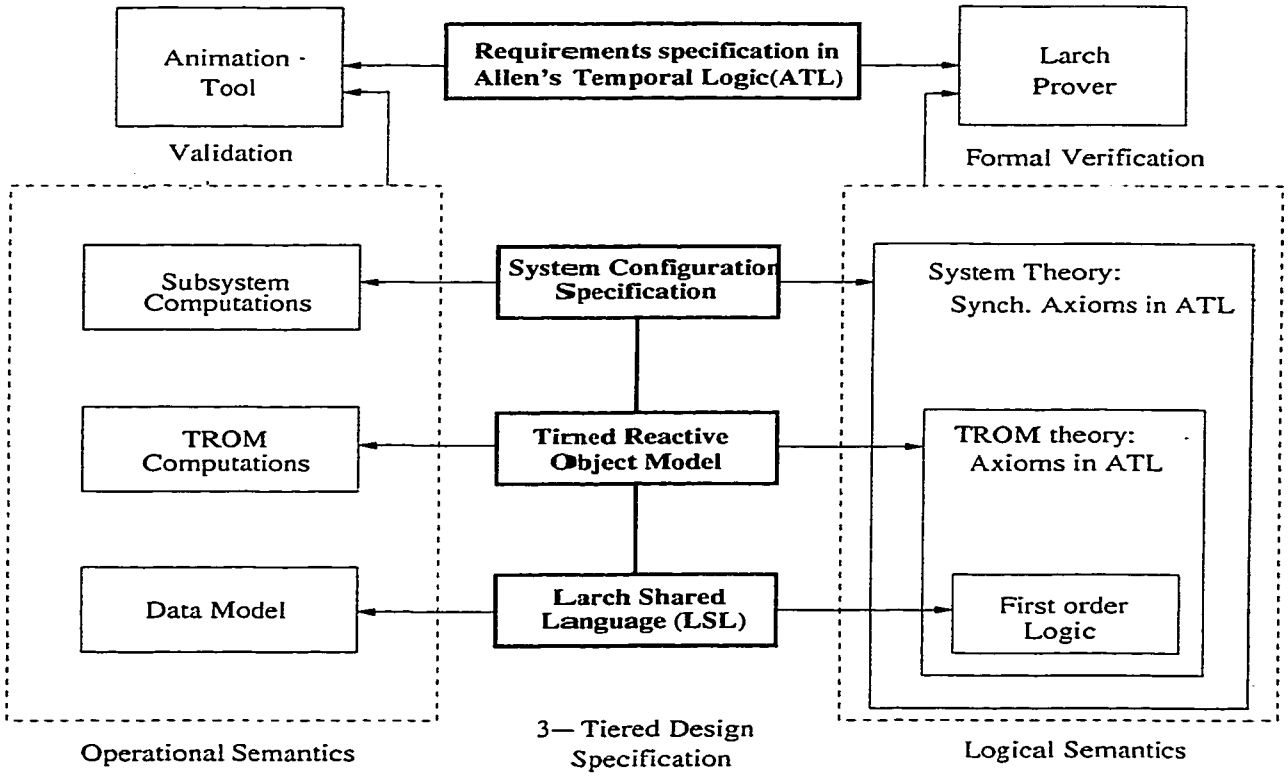

Figure 2: Three Tier

2.1.1 Data Abstraction Tier

The abstract data types included in the class definition of the middle tier are specified in the data abstraction tier. An abstract data type is defined as a Larch Shared Language (LSL) trait. Larch [GH93] provides a two tier approach to specification:

- The Larch Interface Language (LIL) describes the semantics of a program module.
- The Larch Shared Language (LSL) specifies mathematical abstractions referred to in any LIL specification.

Presently, the implementation of TROMLAB includes only LSL traits. Figure 3 shows the LSL trait for set data type.

```
Trait: Set(e, S)
    Includes: Integer, Boolean
    Introduces:
        creat  :     -> S;
        insert : e, S -> S;
        delete : e, S -> S;
        size   : S    -> Int;
        member : e, S -> Bool;
        isEmpty : S   -> Bool;
    Asserts:
end
```

Figure 3: Set trait

2.1.2 TROMTier

A TROM models a *Generic Reactive Class* (GRC). A GRC is an augmented finite state machine with port types, attributes, hierarchical states, events triggering transitions and future events constrained by strict time bounds. A state is an abstraction denoting an environment information or a system information during a certain interval of time. A port type determines the messages that are allowed at a port of that type. All ports of the same type can receive or send messages defined by their port type. A TROM can have several port types associated with it and several ports of the same port type. Two port types are *compatible* if the set of input (output) messages defined by one port type is equal to the set of output (input) messages defined by the

other port type. If two port types are compatible, ports of one type are compatible to ports of another type. An event represents an instantaneous activity, while an action represents an activity taking a non-atomic time interval of finite duration. All events occur at ports. The specification of a transition states the conditions under which an event may occur, and the consequences of such an occurrence. An event might trigger time constrained future events and may result in some computation. The time constraints enumerate the events triggered by a transition and the time bounds within which such events should occur. Thus, a GRC is a class parameterized with port types, and encapsulates the behavior of all TROM objects that can be instantiated from it. In summary, TROM has the following elements:

- A set of events partitioned in three sets: input, output, and internal events.
- A set of states: A state can have substates.
- A set of typed attributes: The attributes can be one of the following:
  - abstract data types,
  - port reference type.
- An attribute function defining the association of attributes to states.
- A set of transition specification: Each specification describes the computational step associated with the occurrence of an event. The transition specification has three assertions: a pre- and post-condition as in Hoare logic, and the port-condition specifying the port at which the event can occur.
- A set of time-constraints: Each time constraint specifies the reaction associated with a transition. A reaction can fire an output or an internal event within a defined time period. Associated with a reaction is a set of disabling states. An enabled reaction is disabled when an object enters any of the disabling states of the reaction.

A formal definition, as given by Achuthan [Ach95], is as follows:
A generic reactive object is an 8-tuple *($\mathcal{P}$, $\mathcal{E}$, Θ, $\mathcal{X}$, $\mathcal{L}$, Φ, Λ, Υ) such that:*

- *$\mathcal{P}$ is a finite set of port-types. 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 two disjoint subsets:* $\mathcal{E}_{ext}$ *is the set of external events, and* $\mathcal{E}_{int}$ *is the set of internal events. The set* $\mathcal{E}_{in} = \{e? \mid e \in \mathcal{E}_{ext}\}$ *is the set of input events, and the set* $\mathcal{E}_{out} = \{e! \mid e \in \mathcal{E}_{ext}\}$ *is the set of output 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 simple, if* $\Phi_s(\theta) = \emptyset$. *By definition, the initial state* $\theta_\circ$ *is atomic.*

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

  - $e(\varphi_{port})$ *where event* $e \in \mathcal{E}$ *labels the transition;* φ_{port} *is an assertion on a reserved variable* pid. pid *is 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 assumed to occur at the null-port* $\circ$.

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

- Υ *is a finite set of time-constraints of the form* $(\lambda_i, e'_i, [l, u], \Theta_i)$ *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, and*
  - $\Theta_i \subseteq \Theta$ *is the set of states wherein the timing constraint* υ_i *will be ignored.*

The language for describing a generic reactive class, derived directly from the formal definition, is shown in Figure 5.

2.1.3 Operational Semantics

The following factors determine a well-formed (syntactically correct) TROM:

- At least one transition leaves every state, thus forbidding a final terminating state.
- If there is more than one transition leaving a state, then the enabling conditions of transitions should be mutually exclusive.
- Before a TROM starts executing, the value of only the active attributes in the initial state are specified. An attribute will acquire a value only when it reaches the first state in which it is active.
- Every computational step in a TROM results in some computation of the TROM.

The anatomy of a TROM shown in Figure 4 describes the dynamic behavior of a well-formed generic reactive object. A stimulus from the environment is received at a port if the port condition can be satisfied by the stimulus. The stimulus received in the form of an input event at a particular state should satisfy one of the enabling conditions in that state; otherwise, the stimulus is responded with a message and no further effect occurs. If the enabling condition is satisfied, the transition associated with the enabling condition happens. The state change may involve some computation at the destination state. Due to the transition, some time constrained reactions might be triggered, or the system may progress by performing an internal action, or some response to the environment may be given in the form of an output event. A description of the dynamic behavior of a TROM object is given below.

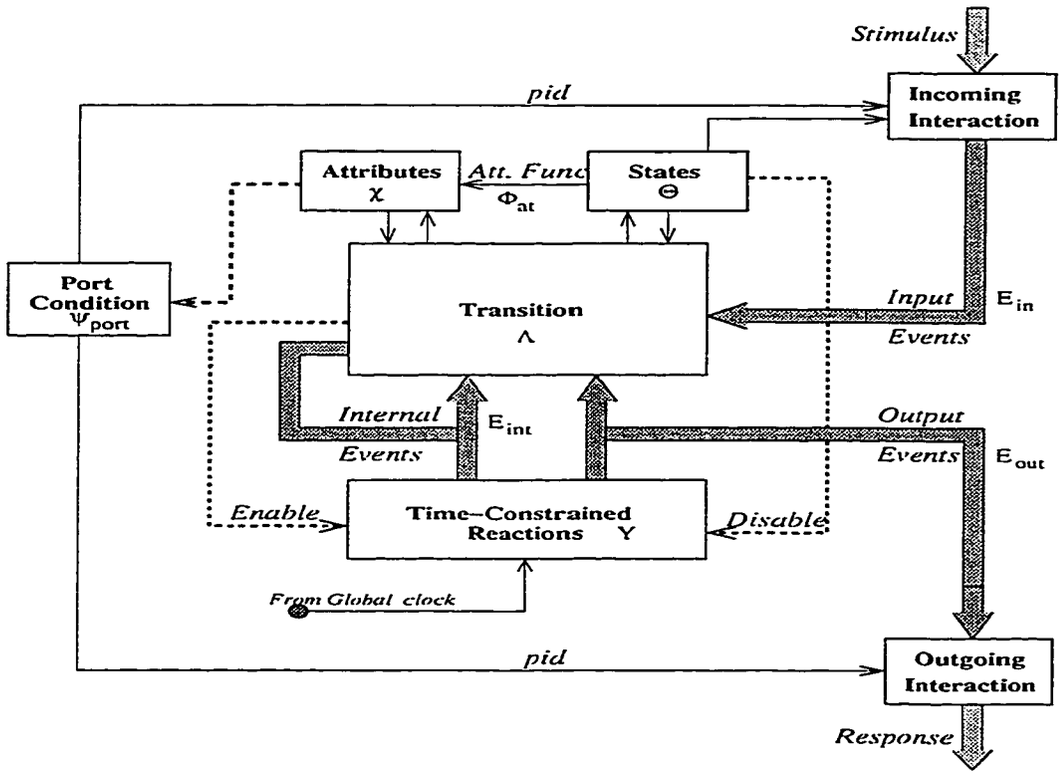

Figure 4: TROM class anatomy

A message from an object to another object in the system is called a *signal* and is represented by a tuple $\langle e, p_i, t \rangle$, denoting that the event e occurs at time t, at a port p_i. The *status* of a TROM at any time t, is the tuple $(\theta; \vec{a}; \mathcal{R})$, where the current state θ is a simple state of the TROM , $\vec{a}$ is the assignment vector, and $\mathcal{R}$ is the vector of outstanding reactions.

The reaction vector associates the set of reaction windows with each time constraint, where a reaction window represents an outstanding timing requirement to be satisfied by the output event or the internal event associated with the time constraint. When the reaction vector is null the TROM is in a stable status.

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

A computational step of a TROM is an atomic step which takes the TROM from one status to its succeeding status as defined by the transition specifications. Every

computational step of a TROM is associated with the transition of the TROM and every transition with either an interaction signal or an internal signal or a silent signal. The computational step occurs when the TROM receives a signal and there exists a transition specification such that the following conditions are satisfied: the triggering event for the transition is the event causing the signal; the TROM is in the source state or in a substate of a source state of transition specification; the port condition is satisfied if the signal is in the interaction and the enabling condition is satisfied by the assignment vector. The effects of the computational step are: the TROM enters the destination state; the assignment vector is modified to satisfy the post condition; and the reaction vector is modified to reflect the firing, disabling, and enabling of reactions. Each computational step is associated with the transition in the state machine of the TROM. That is, when the object with status $(\theta; \vec{a}; \mathcal{R})$, receives a signal $\langle e, p_i, t \rangle$ and there exists a transition specification that can change the status of the TROM a computational step occurs. A computation c of a TROM object $\mathcal{A}$ is a sequence, possibly infinite, of alternating statuses and signals, $\mathcal{OS}_0 \stackrel{\langle e_0, p_0, t_0 \rangle}{\rightarrow} \mathcal{OS}_1 \stackrel{\langle e_1, p_1, t_1 \rangle}{\rightarrow} \ldots$. Two objects that communicate through events $e?$ and $e!$ *synchronize* their transitions labelled by these events.

After the transition is taken the current state will be the destination state of the transition. The port at which the interaction must satisfy the port condition associated with the transition, thereby constraining the objects with which the TROM can interact at that instance.

A computational step causes time-constrained responses to be activated or deactivated. If the constraint event of the outstanding reaction is the event associated with the transition, and the time of occurrence of the event associated with the transition is within the reaction window of the outstanding reaction, then the reaction is fired. If the destination state of the transition associated with the computational step is a disabling state for an outstanding reaction then the reaction is disabled. Whenever a reaction is time-constraint by the transition associated with the computational step, the reaction is enabled. The operational semantics ensures that the time cannot advance past reaction window without either firing or disabling the associated outstanding reaction.

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

Figure 5: Template for TROM Class Specification.

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

Figure 6: Template for System Configuration Specification.

2.1.4 Subsystem Tier

In the subsystem tier a subsystem configuration is specified. This specification uses objects instantiated from classes specified in the second tier. An object is instantiated from a class by creating a finite number of ports for each port type in the class specification, and by initializing the attributes included in the class. Each instantiated object will carry its own set of attributes. A port link is an abstraction of a communication medium between two objects. A port link is established between a port of one object and a compatible port in another object. Objects communicate by exchanging messages (external events) through the port links. The syntax for subsystem specification is shown in Figure 6. The Include section lists imported subsystems. A reactive object is created in the Instantiate section, with parametric substitutions to cardinality of ports for each port type. The Configure section defines a configuration obtained by composing objects specified in the Instantiate section and in the subsystem specifications imported through the Include section. The composition operator $\leftrightarrow$ sets up communication links between compatible ports of interacting objects. Two ports are compatible if the set of input message sequences at one port is a subset of the output message sequences at the other port. For example, a link $A_i.@c_j \leftrightarrow B_k.@p_l$ in the Configure clause means that the port c_j of object A_i and the port p_l of object B_k are connected in the subsystem. The port links effectively determine the set of all valid messages that can be exchanged among the objects in a subsystem.

A *computational step* of a TROM occurs when the object with status $(\theta; \vec{a}; \mathcal{R})$,

receives a signal $\langle e, p_i, t \rangle$ and there exists a transition specification that can change the status of the TROM . Two objects that communicate through events $e?$ and $e!$ *synchronize* their transitions labelled by these events. The state machine, that gives the behavior of the subsystem, is the synchronous product of the state machines corresponding to the instantiated objects and included subsystems in the SCS specification. The computation of a subsystem is an infinite sequence of system statuses and signals that affect status changes. Although a reactive object has no internal parallelism, concurrent computations are possible within a subsystem.

2.2 TROMLAB Components

The TROMLAB framework includes a Rose interface and a graphical user interface. The former is used to create visual models of the reactive system components. The latter is used to interact with the rest of the TROMLAB components. In this section we briefly review the functionality of the *Translator* [Pop99], *Interpreter* [Sri99], and the *Simulator* [Hai99].

2.2.1 The Rose-GRC Translator

The translator is implemented by Popistas [Pop99], using Rose script, a scripting language supported by Rational Rose [Rat98]. The translator is a tool to automatically translate the graphically designed models: class, state chart, collaboration, and sequence diagrams into TROM formal specifications. The diagrams are modeled using Rational Rose that supports UML based system modeling. The translator takes the Rose model as an input and produces text files: TROM class formal specifications, message sequences, and subsystem configuration.

The Rose-GRC translator consists of the following parts:

- *Interface to Rose:* It is the user interface that provides access to create visual representations and invoke the translator.
- *Translator:* Translator takes the specified Rose diagrams and performs the following tasks:
  - Checks the correctness of the Rose diagrams by performing syntactic and semantic check to ensure that the models conform to TROM formalism.

- Handles any error occurring during the execution by producing clear and specific error messages.
- Translates the Rose diagrams into an internal structure, using record data types.
- Produces the textual specifications according to the syntax presented in the previous section 2.1.

Figure 7 shows the architecture of the Rose-GRC translator, as proposed by [Popa1999].

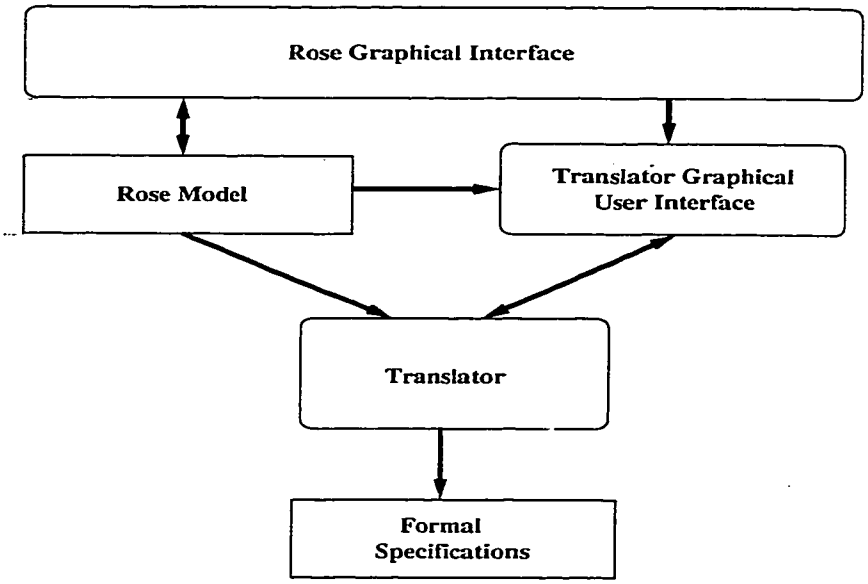

Figure 7: GRC-Translator Architecture

2.2.2 The Interpreter

The interpreter was the first tool to be implemented in TROMLAB by Tao [Tao96]. This implementation, in C++, required the textual descriptions of all the three levels to be provided as a single source file. The advantage of the three-tiered design was not realized in this implementation. Haidar [Hai99] and Sriniva [Sri99] reengineered the interpreter implementation in Java. This version included incremental and independent compilation of specifications, and enhanced error reporting. The interpreter checks the syntactic correctness of specifications and builds an internal representation of the well-formed formal specification of a reactive system. Figure 8 shows the

architecture of the Interpreter.

In order to build the internal representation it performs the following tasks:

- *Syntactic analyser:* It makes sure that the files are syntactically correct; that is, consistent with TROM grammar.
- *Semantic analysis:* It does simple semantic analysis such as
  - states of a TROM have different names,
  - an LSL trait is used after being declared,
  - every transition has an outgoing and incoming state, and
  - transition specifications are well-formed logical formulas
- *Internal structure:* Based on a syntactically and semantically correct text file it generates an Abstract Syntax Tree, an internal representation for the models, that would be used by all the other tools in TROMLAB.

The components of interpreter are the following:

- *Parsers*
  The parsers are implemented in JavaCC and JJTree. Java Compiler Compiler is a parser generator for use with Java applications that produces Java code. JJTree is a preprocessor for JavaCC that inserts in JavaCC source actions for parse tree building. There exist separate parsers for *LSL trait,* TROM *class specification, SCS,* and *initial simulation event list* parsers.
- *Semantic analyzer*
  Semantic analysis is done in two phases: in phase 1, semantic analysis internal to a class specification is done; and in phase 2, semantic analysis of the subsystem configuration is done.

2.2.3 The simulator

An animation tool was implemented by Muthiayen [Mut96] which worked with the first interpreter. The reengineered simulator tool, designed and implemented by Haidar [Hai99], works with the new interpreter and has reasoning capabilities. Figure 9 shows the simulator architecture. The simulator can work in one of two modes:

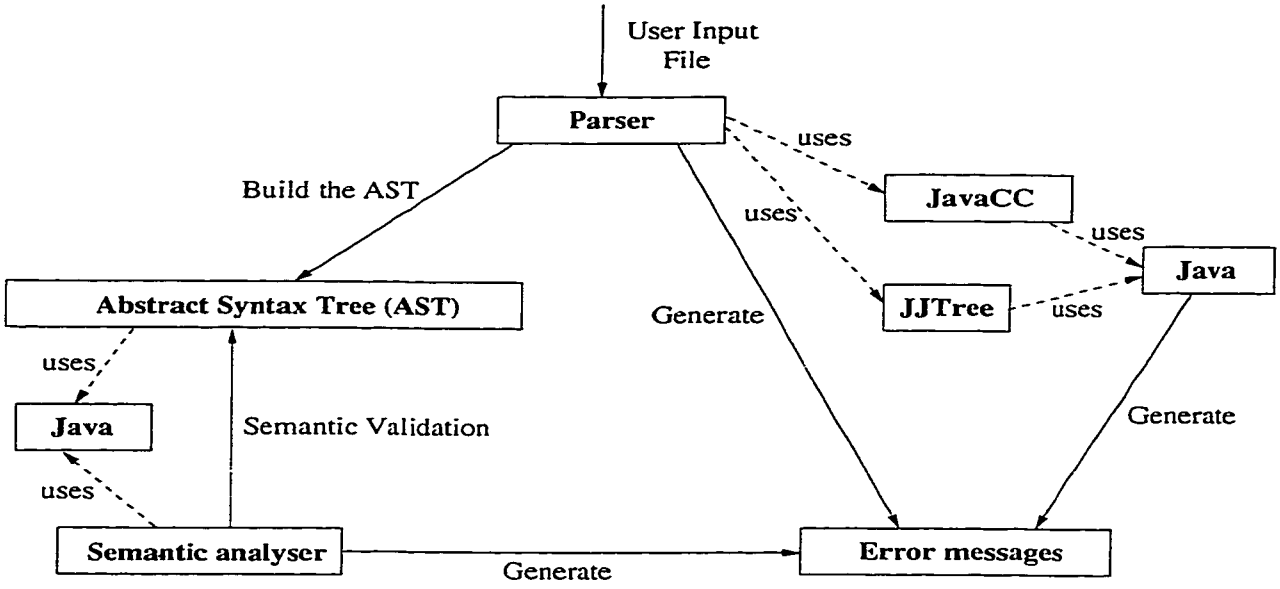

Figure 8: Architecture of interpreter

- *Debugger mode:* In this mode the developer can, at the end of every handled event, invoke the debugger and use it to query the system. The system can be rolled back and new events can be injected.

- *Normal mode:* In this mode the simulation will go on uninterrupted until the system goes into a stable state. The result of a simulation is one scenario of what could happen, given the initial set of events.

The simulation tool consists of the following components:

- *Simulator* It consists of an event handler, a reaction window manager, and an event scheduler.

  - The event handler is responsible for handling the events which are due to occur and detects the transition which the event will trigger.

  - The reaction window manager is responsible in activating the computational step to handle the transition causing events to be fired, disabled or enabled.

  - The event scheduler causes an enabled event to occur at a random time within the corresponding reaction window. It schedules output events through the least recently used port using a round robin algorithm.

- *Consistency checker* It ensures the continuous flow of interactions by detecting deadlock configurations.

- *Validation tool* It consists of a debugger, a trace analyzer, and a query handler.

  - The debugger supports system experimentation by allowing the user to examine the evolution of the status of the system throughout the simulation process. It also supports interactive injection of simulation event, and simulation rollback to a specific point in time.

  - The trace analyzer includes facilities for the analysis of the simulation scenario. It gives feedback on the evolution of the status of the objects in the system, and the outcome of the simulation event.

  - The query handler allows examining the data in the AST for the TROM class to which the object belongs, and supporting analysis of the static components during simulation.

- *Object model support* It supports the specification of the TROM classes and the evaluation of the logical assertions included in the transition specifications.

- *Subsystem model support* It creates subsystems by instantiating included subsystems, with its objects and port links.

- *Time manager* It maintains the simulation clock and updates it regularly. It allows setting the pace of the clock to suit the needs of analysis of simulation scenarios. It also allows freezing the clock while analyzing the consequences of a computation.

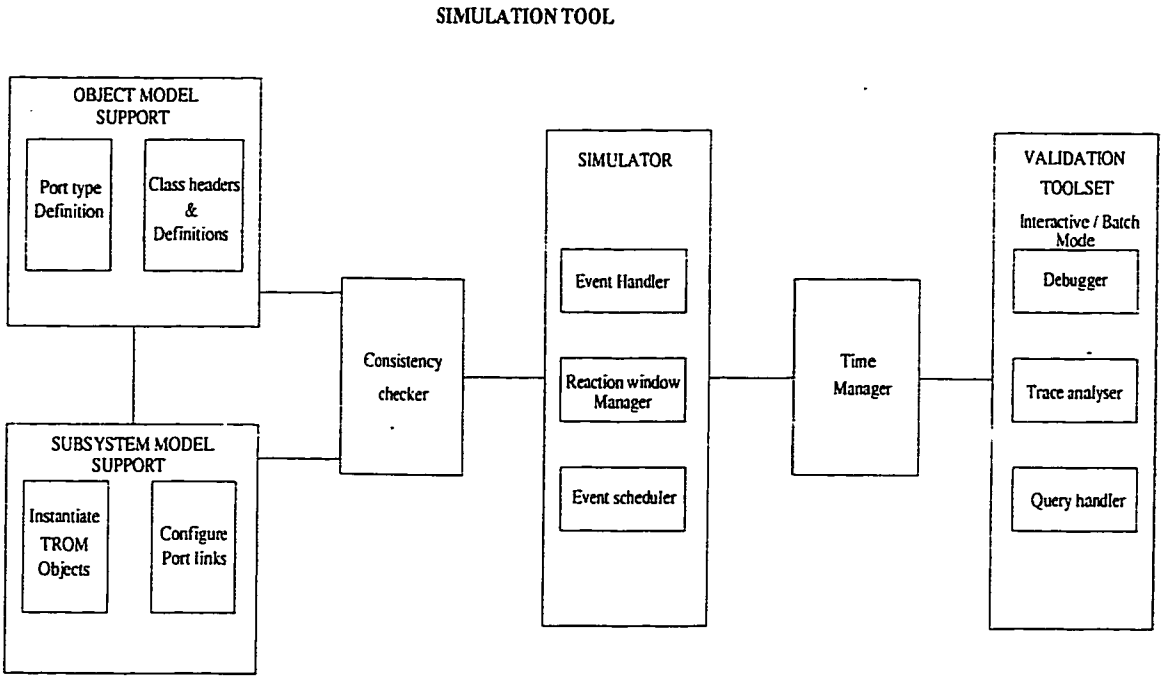

Figure 9: Architecture of simulation tool

Chapter 3

Parameterized Events

In this chapter, we show how to augment the TROM formalism to accommodate parameters in the representation of events. We describe the syntactic and semantic extensions of TROM formalism.

3.1 Introducing Event Parameters to TROM

In several applications, such as mobile telephone and protocol engineering, a large number of system and environmental events have similar functionalities. The only difference may be in the data values to be exchanged between components affected by these events. For large systems the OO models may have too many classes, and several hundred objects may be involved in defining a subsystem. The state machine models for an object may also become too complex to comprehend due to the explosion in the number of states and events. The common functionalities of events in such systems can be the basis of a solution that contains the complexity. In fact, the representation of events with common functionalities can be made more expressive when parameters are used. The use of parameters in TROM events is analogous to the use of function parameters in programming languages or arguments for a predicate. A parameterized event represents a family of events that are distinguished by the (dynamic or static) values that can be assigned to its parameters.

Introducing parameterized events to TROM achieves three goals:

- Parameters reduce the complexity of comprehension and avoid the state explosion of a system. Figure 10 (a) shows an example of a state diagram with

three different transitions having three triggering events *(E1, E2, E3)* leading to three different states *(S1, S2, S3)*. These events depict the same environmental action applied on one environmental attribute with different values. Their common feature is that the same precondition P implies the same post condition Q. Now, this situation in large systems causes state and event explosions. We avoid this situation by introducing the parameterized version of this example, as shown in Figure 10 (b); we introduce, for the transitions having the same pre and post conditions, a parameter denoting the environmental attribute, and carrying it withing a paramter list that is followed by the event. Therefore, the states *(S0, S1, S3)* are replaced by one state *S1*, the number of events are reduced to one event E, and the number of transitions to one transition.

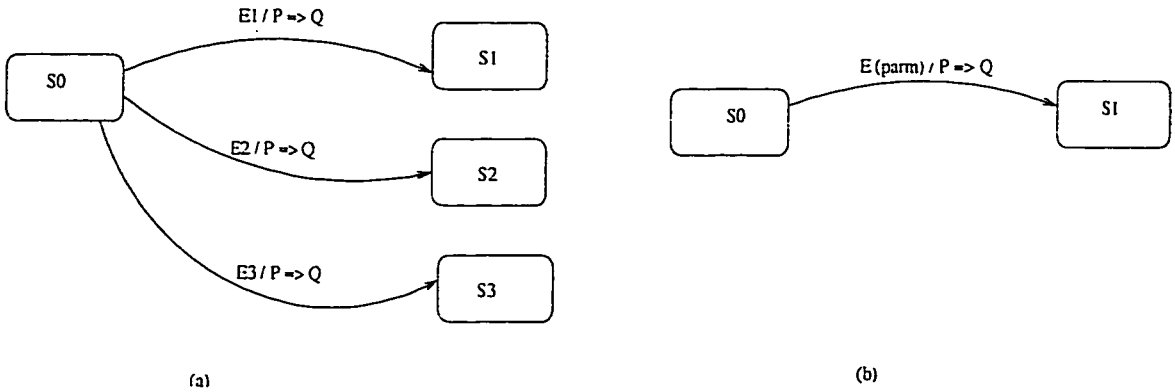

Figure 10: State diagrams (a) without and (b) with parameterized events

- Parameters add expressiveness to TROM formalism. Figure 11(a) shows a state diagram with three transitions having three different events *(E1, E2, E3)*. Each transition has a different post condition. The three events, as the example given above, represent one environmental action applied on an environmental attribute. The resulting post conditions differ relatively with the different values that this attribute can carry. Therefore, to improve the expressiveness of the behavior of the object, we introduce a parameter that represents the related environmental attribute. This parameter can be carried, with the different values that it can have, within the parameter list of the event. This way, it is more clear and expressive to have the same event carrying different values of the same parameter. Figure 11(b) shows events *E1, E2, and E3* reduced to one event E parameterized with the parameter *parm*, and in each transition, *parm*

has a different value.

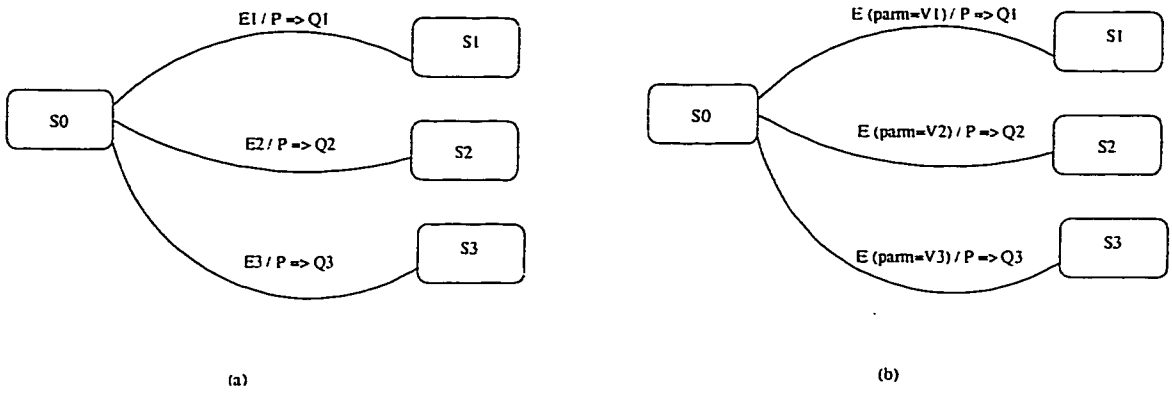

Figure 11: State diagrams (a) without and (b) with parameterized events

- Parameters can be used to transport data between objects in a subsystem.

However, in order to parameterize the TROM representation of events, it is crucial to extend the formalism of TROM in order to set the rules and grammar for parameterized events.

3.2 Extending TROM Formalism

3.2.1 Syntax of Parameterized Events

The formal definition of TROM presented in Chapter 2, will be modified as follows after integrating the formal definition of parameters. The syntax of parameterized events can be directly extracted from the modified TROM formal definition:

A generic reactive object is a 9-tuple $(\mathcal{P}, \mathcal{E}, \Theta, \mathcal{X}, \mathcal{Y}, \mathcal{L}, \Phi, \Lambda, \Upsilon)$ *such that:*

- $\mathcal{P}$ *is a finite set of port-types. 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 two disjoint subsets:* $\mathcal{E}_{ext}$ *is the set of external events, and* $\mathcal{E}_{int}$ *is the set of internal events. The set* $\mathcal{E}_{in} = \{e? \mid e \in \mathcal{E}_{ext}\}$ *is the set of input events, and the set* $\mathcal{E}_{out} = \{e! \mid e \in \mathcal{E}_{ext}\}$ *is the set of output 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.*

- $\mathcal{Y}$ *is a finite set of typed parameters. The set* $\mathcal{Y}$ *can be partitioned into two disjoint subsets:* $\mathcal{Y}_d$ *is the set of dynamic parameters, and* $\mathcal{Y}_s$ *is the set of static parameters.*

- $\mathcal{L}$ *is a finite set of LSL traits introducing the abstract data types used in* $\mathcal{X}$.

- Φ *is a function-vector* $(\Phi_s, \Phi_{at}, \Phi_p)$ *where,*

  - $\Phi_s : \Theta \rightarrow 2^{\Theta}$ *associates with each state* θ *a set of states, possibly empty, called substates. A state* θ *is simple, if* $\Phi_s(\theta) = \emptyset$. *By definition, the initial state* θ_o *is atomic.*

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

  - $\Phi_p : \Theta \rightarrow 2^{\mathcal{Y}}$ *associates with each state* θ *a set of parameters, possibly empty, called active parameter set. At each state* θ, *the set* $\overline{\Phi_p} = \mathcal{Y} - \Phi_p(\theta)$ *is called the dormant parameter 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 :* $< \langle \theta, \theta' \rangle ; e[\varphi_{parm}](\varphi_{port}); \varphi_{en} \Longrightarrow \varphi_{post} >$; *where:*

  - $\theta, \theta' \in \Theta$ *are the source and destination states of the transition;*

  - $e[\varphi_{parm}](\varphi_{port})$ *where event* $e \in \mathcal{E}$ *labels the transition;* φ_{parm} *is an assertion on the parameters in* $\mathcal{Y}_d$ *that can be absent if* $\mathcal{Y}_d$ *is empty;* φ_{port} *is an assertion on a reserved variable* pid. pid *is 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 assumed to occur at the null-port* o.

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

- Υ *is a finite set of time-constraints of the form* $(\lambda_i, e_i', [l, u], \Theta_i)$ *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, and*
  - $\Theta_i \subseteq \Theta$ *is the set of states wherein the timing constraint* υ_i *will be ignored.*

After modifying the definition of TROM, the anatomy of TROM class shown in Chapter 2 is changed respectively as seen in Figure 12.

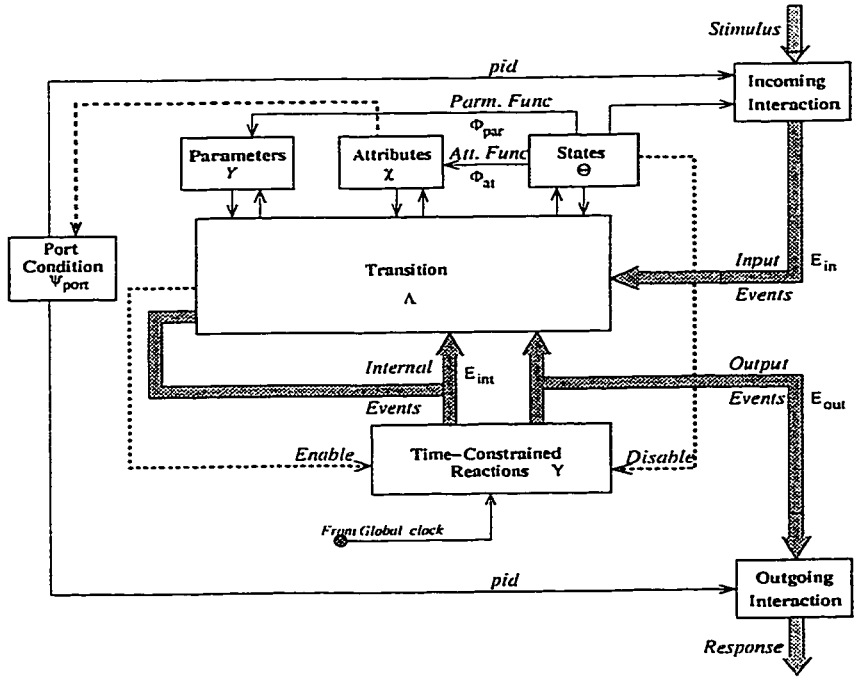

Figure 12: New TROM class anatomy

Figure 13 shows the new template for the TROM class specification with parameters. It includes a new section **Parameter–Specifications.**

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

Figure 13: Extended template for a class specification

Here, we should mention that the syntax of the Subsystem Configuration of TROMs in a system is not affected by the new TROM syntax of parameterized events. Only the syntax of a TROM class is modified. However, the objects instantiated from TROM classes of a system carry with them the parameters defined in the respective classes and can independently operate on them.

3.2.2 Semantics of Parameterized Events

In TROM formalism, parameters are globally defined and hence are common to all TROM classes. In this sense, parameters are similar to abstract data types defined in the lowest tier of TROM formalism. Parameters are of two kinds: *constant* and *variable*. A constant parameter can be a constant of any type: *integer*, *char*, or *string*. The value of a constant parameter cannot be changed by any object in the system. The value of a variable parameter can be changed by any object in the system, and the updated value is inherited by all other objects. For this reason, parameters are to be declared in all the classes that benefit from their usage since TROM formalism does not support global declarations or data sharing.

Both constant and variable parameters behave like attributes; however, there are subtle differences. These differences are mainly expressed in two aspects. The first is in the way parameters are associated with states of an object. For each state, the parameter function defines a subset of parameters that are active in the state. Parameters to be associated with the state are identified from incoming and outgoing transitions of the state. Like attributes, the parameters that appear in the post condition of the incoming transitions are associated with the state. However, the parameters that

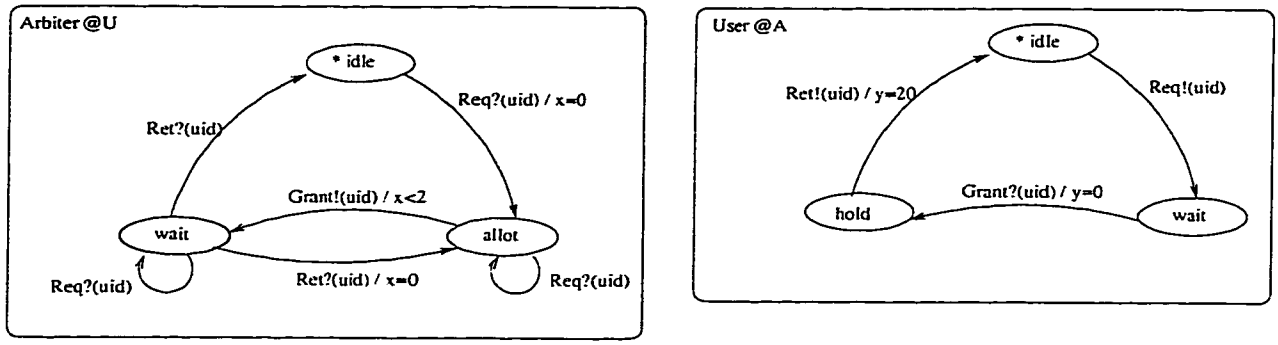

Figure 14: Arbiter-User Example

appear in the parameter list of the outgoing transitions are also associated with the state. Second, the usage of the parameters throughout the behavior of the TROM is different from the behavior of attributes. Not only parameters appear in the enabling condition and the post condition of a transition, but also a parameter list is identified for each transition. It is a new element added to the transition specification. This list appears after the triggering event as an assertion on the variable parameters. Also, constant parameters are used to complete an assertion or a computation on a variable parameter.

We illustrate the use of parameters through a brief example of an arbiter that allows several processes to use a single resource. The arbiter allocates the resource, based on its availability, to the requesting process. In this example, we only focus on the parameterization issue. Figure 14 shows the state machines that describe the behavior of the two classes: *Arbiter* and *User*. When a *User* requests the resource, the *Arbiter* grants it access to the requested resource within a certain time bound. The *User* is allowed to access the resource for a limited period of time before releasing it. On the other hand, if, at the time of the request, the resource is already allocated, the *User* has to wait until the resource is released.

In order to request a resource, the *User* sends to the *Arbiter* the message *Req* with the parameter *uid*, which is the id of the *User*. This means that both the *Arbiter* and the *User* synchronize between each other using the parameter *uid* along with the exchanged message. Notice that the same *uid* appears in the output event of the *User* and in the input event of the *Arbiter*. This applies to all the other events as well. On the other hand, the *uid* parameter is associated with the source states of these

transitions in which it appears. For instance, in class *User*, the *uid* is associated with the *idle* state. Also, the value of the *uid* parameter, as seen by the two classes is the same at any instant.

In the next chapter we describe how the TROM template and the TROMLAB tools are modified to include and manage parameterized events.

Chapter 4

Parameterized Events - TROM Formalism Redefined

To accommodate parameterized events into TROM we realized the necessity to redefine TROM notation and reengineer its environment tools. Redefining TROM notation was achieved by keeping in mind two basic guidelines:

- TROM class specifications should stay simple; and
- syntactic and semantic changes should only be a conservative extension to the original TROM notation.

We arrived at these guidelines to ensure that only minor changes will be required to reengineer the tools in TROMLAB framework. In other words, we want to maximize code reuse.

In this chapter, we discuss the extensions to Real-Time UML (RTUML) modeling notation and the changes to Rose-GRC translator in order to mechanically generate TROM formal specifications according to the revised syntax. We also discuss the reengineering of the Interpreter. We illustrate parameterized events in the modeling and specification of Train-Gate-Controller example, a bench-mark problem studied by the real-time research community.

4.1 Redefining TROM formal Representation

Here, we describe how to redefine the formal specification of TROM to include the parameter representation. The redefinition represents changes and additions to the existing grammar as well as modifications to the structure of the TROM class formal template itself. In the following we explain the existing TROM formal representation grammar.

4.1.1 TROM Formal Specification without Parameters

A TROM class representation includes (see Table 1): the class title, events, states, attributes, LSL traits, the attribute functions, the parameter specifications, the transition specifications, and the time constraints.

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

Table 1: Grammar for generic reactive class specification

According to the TROM grammar:

- A class title (see Table 2) is described by the keyword Class, followed by a string denoting the class name, the it is followed by a list of port types included in square brackets. The port type list consists of one or several port type names separated by commas and each of which is a string prefixed by the symbol @.

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 2: Grammar for generic reactive class title

- Event (see Table 3) representation starts by the keyword Events:, followed by a list of events that can include one or several events separated by commas. An event can be one of three kinds: internal event, input event, or output event. An event is denoted by a string representing the event name. However, an input event, respectively output event, name is followed by the character ?,

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 3: Grammar for events

respectively !, followed by a string representing the port type at which the event occurs.

- A state (see Table 4) representation starts by the keyword States:, followed by a set of states. The state set representation consists of the initial state proceeded by the character *, followed by one or several states separated by commas. A state can be either a string denoting a simple state name, or if it is a complex state, it is represented by a string as the state name followed by a state set included between curly brackets.

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 4: Grammar for states

- Attributes (see Table 5) representation starts by the keyword Attributes:, followed by a list of attributes. The attribute list consists of one or several attributes separated by commas. An attribute is denoted by a string followed by a colon followed by the attribute type. The attribute type can be a port type name proceeded by the character @, a LSL trait name, or a simple data type represented by the keywords Integer or Boolean.

- LSL traits (see Table 6) representation starts by the keyword Traits:, followed by a list of traits. The list of traits consists of one or several traits. A trait is

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 5: Grammar for attributes

denoted by a string as the trait name followed by an argument list and a string denoting the trait type both included between square brackets. An argument list consists of one or several arguments. An argument is a string denoting a trait type name or a port type name prefixed by the character @.

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 6: Grammar for LSL traits

- Attribute functions (see Table 7) representation starts by the keyword Attribute-Functions:, followed by a list of one or several attribute functions separated by semicolons. An attribute function consists of a string denoting a state name followed by the keyword ->, followed by an attribute list included between curly brackets. An attribute list consists of zero or more strings denoting attribute names separated by commas.

- Transition specifications (see Table 8) representation starts with the keyword Transition-Specifications, followed by a list of transition specifications separated by semi-colons and new lines. A list of transition specifications consists of one or several transition specifications. A new line indicates the beginning of a new

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 7: Grammar for attribute functions

specification. Each transition specification includes: a string that represents the name, a colon, one or several state pairs separated by semi-colons, a triggering event, an assertion, the implication operator (r), and another assertion. A state pair consists of two string state names between brackets and are separated by a comma. A triggering event is a string event name followed by an assertion between brackets. An assertion is a either a 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 — between them. A term is either a factor or two factors with the logical operator & between them. A factor is one of the following: a factor proceeded by the logical operator !, the reserved variable pid, a primed attribute, an attribute, logical expressions true or false, an LSL term, or an assertion in brackets. A LSL term consists of a string LSL function name, followed by an argument list in brackets. An argument list includes one or several arguments. An argument can be one of three: the reserved variable pid, a string attribute name, or a LSL term. A primed attribute is an attribute (from the attribute function) followed by the character '.

- Time constraints (see Table 9) are represented by the keyword Time-Constraints:, followed by one or several constraints separated by semi-colons and new lines. A constraint consists of a string name followed by a colon and the string name of the constraining transition specification, the string name of the constraining event, the lower and upper bounds, and a list of disabling states. The lower and upper bounds are represented by naturals and are proceeded and followed by open and closed interval indicators. The list of disabling states consists of zero or several string state names separated by commas.

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 8: Grammar for transition specifications

4.1.2 TROM Formal Specification with Parameters

We describe the necessary modifications and additions to TROM formal specification to include the representation of parameters while preserving the readability and understandability of the resulting TROM specification.

Introducing parameters necessitates the following assumptions and changes to the grammar:

- Parameters are declared in the Attributes section and syntactically treated as attributes.
- Like attributes, parameters are associated with states. Therefore, they are assigned to states through the parameter functions and appear in the Attribute-Functions section.

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 9: Grammar for time constraints

- A new section is included, Parameter–Specifications (see Table 10). It can have zero or several entries. An entry consists of a string event name followed by a list of parameters. This list represents all possible parameters associated with the named event and it consists of one or several parameters separated by commas. A parameter is a string representing its name.

Parm–specs	::=	Parameter–Specifications : NL < Parameter–Section >
Parameter–Section	::=	EMPTY | <Entry> <More-entries>
Entry	::=	<event–name> ; <parm–name> <Parm–list> NL
More–entries	::=	EMPTY | <Entry> <More–entries>
Parm–list	::=	EMPTY | , <parm-name> <Parm-list>
event–name	::=	String
parm–name	::=	String

Table 10: Grammar for parameter specifications

- Transition specifications section is extended (see Table 11) to include the triggering event argument list where parameters along with their values are revealed. The grammar of the Transition-Specifications section is changed as follows: the triggering event becomes an event name followed by a parameter list followed by an assertion between brackets. An event name is a string denoting the event name. The parameter list consists of zero or several parameter entries separated by commas and all included between square brackets. A parameter entry is one of two: a parameter name, or a parameter and a value with the logical operator

= between them. A parameter is a string denoting the parameter name. A value is a natural, an integer, a string, or a Boolean.

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> <Parm_list> (<assertion>)
Parm_list	::=	Empty | [<Parm_entry> <More_parm_entries>]
Parm_entry	::=	<parm_name> | <parm_name> = <value>
More_parm_entries	::=	EMPTY |, <Parm_entry> <More_parm_entries>
Parm_name	::=	String
value	::=	Nat | Integer | String | Boolean
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 11: Modified grammar for transition specifications

4.2 Extending Real Time Model Notation in Rose

We describe the suggested extensions to the Real time modeling notation introduced in [MA98a] and [MA98b]. These extensions allow visual modeling of parameters in Rational Rose for UML.

4.2.1 Rose Model Notation without Parameters

In Rational Rose, the model notation for reactive classes depicts modeling the class diagrams, state diagrams, sequence diagrams, and collaboration diagrams. We present a brief description of the rose model notation. For detailed information see [Pop99].

Class diagram Class diagrams are the logical static view of the model. The modeling notation for a GRC is shown in Figure 15. A class can be of two types a GRC or a PortType. A PortType class is an aggregate of a GRC. Stereotype modeling element is used to provide sub-classification that has been given more specific meaning. A GRC is known by the stereotype GRC followed by the class name. A PortType is known by the stereotype PortType followed by the port type name. In the class is defined the list of attributes. Attributes also can be of two types distinguished by setting their stereotypes. An attribute stereotype is either a DataType or a PortType. A PortType has one attribute events, which is a set of incoming and outgoing events that occur at this port type.

Figure 15: Rose GRC

Statechart diagrams A statechart diagram is the dynamic behavioral view of the model. It shows the states of a given class and the transitions of the statechart. Every transition has a source state and an end state. A transition consists of a triggering event, a guard condition, and an action. A guard condition is composed of a port-condition for the port at which the event occurs, pre-condition for triggering the event, and a time condition all separated by the logical operator &&. An action consists of a post condition of the triggered event, and a time constraint constraining the event. Figure 16 depicts a statechart.

Collaboration and sequence diagram In Rational Rose a collaboration diagram shows a sequence of interactions between objects. For our purposes, we use only the static aspect that can be captured in a collaboration diagram, which is defining objects and the links between them. A sequence diagram is a type of interaction diagram that shows a sequence of interactions between objects by showing the message

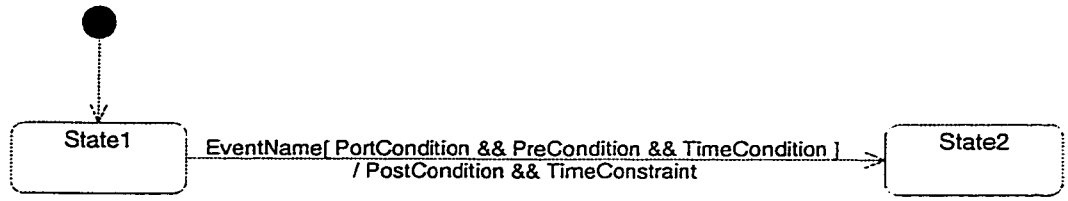

Figure 16: Rose StateChart

passing along with the time line of each object. Figure 17 and Figure 18 depicts a collaboration and a sequence diagrams.

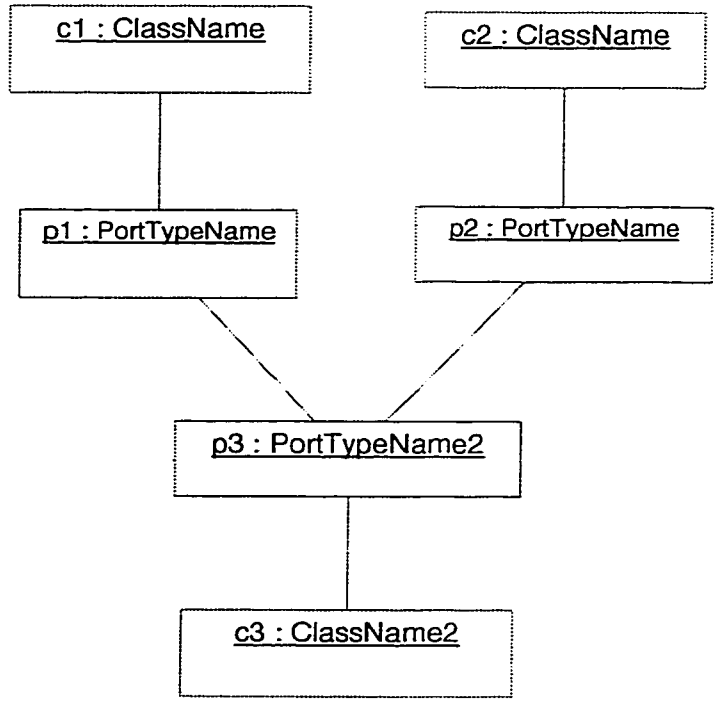

Figure 17: Rose Collaboration Diagram

4.2.2 Changes to Visual Modeling Notation

Modeling reactive systems in Rose requires the following changes:

Parameters Declaration: Parameters are declared in the GRC class in the attribute section. We distinguish between an attribute declaration and a parameter declaration by setting the stereotype of the declared parameter as Parameter. We distinguish static parameters from dynamic ones by checking the static field in the

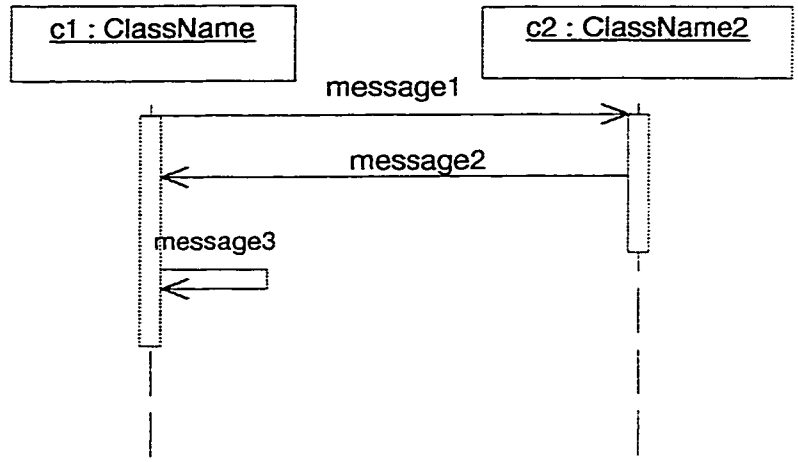

Figure 18: Rose Sequence Diagram

attribute specification window. Thus, the sign $ will appear before the parameter name to denote it as a static parameter (see Figure 19).

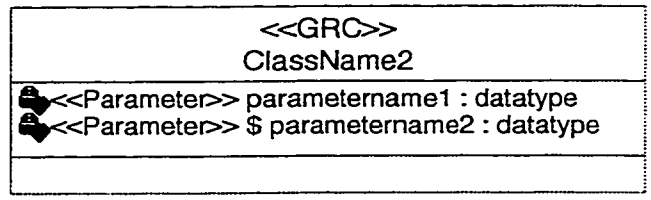

Figure 19: Rose Parameters Declaration

Parameters Usage: As shown in Figure 20. For each transition, the triggering event name is followed by a list of parameters. The representation of parameters can be done from the transition specification window by entering the parameter list in the arguments field. Parameters can appear in two different forms in the event parameter list:

- By reference : where the parameter is referenced by its name. It implies that the firing of the event is not affected by the value of the parameter.
- By value : where the parameter is assigned a specific value from its domain. It implies that the firing of the event depends on the parameter's value. The syntax of event parameters is:
  Event(par1[=val1], par2[=val2], park[=valk])[GuardCond]/ PostCond && Time-Cond

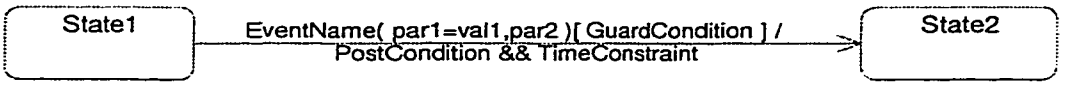

Figure 20: Rose Parameters Representation

4.3 Rose-GRC Translator

In this section, we describe the modifications needed to make the translator conform to the revised TROM grammar while retaining the original design decisions. We also present the old design of the translator and the new design that accommodates parameterized events. The data structure of parameter and the algorithms for the added design implementation are included in Appendix A.

4.3.1 Upgrading the Translator

Here, we discuss the improvements needed to extend the translator model.

- *Parameter declaration:* parameters should be declared as attributes in the attribute section.
- *Parameter-function:* parameters are associated with states. Therefore, a parameter function has to be designed and implemented in the new translator. This function extracts the parameters from transitions and associates them with states as follows. First, the parameters appearing in the argument list of the event of a transition are associated with the source state. Then, the parameters appearing in the post condition of the transition are associated with the destination state. The result of the parameter function is combined with the one of the attribute function. Thus, in the attribute function section, each entry becomes a state followed by a set of combined attributes and parameters.
- *Parameter Usage:* in order to express the behavior of parameters in a TROM, the functionality of the translator should be extended as follows:
  - The translator should extract the parameter lists of the events from the transition representations in Rose and add them to the Transition-Specifications section.

- For all the occurrences of a triggering event, the parameters appearing in the corresponding argument lists are extracted and assigned to this event in the Parameter-Specifications section, which is added to the formal specification.

- *Logical checks:* The translator should ensure correctness in the output corresponding to parameters. Therefore, some logical checks need to be performed in order to guarantee that:

  - No duplication should occur in declaring parameters.
  - No duplicate parameters are associated to a state.
  - Every used parameter is valid and properly declared.
  - The occurrence of parameters in the argument list is syntactically correct according to the grammar presented in Section 4.1.
  - For each state, all the outgoing transitions having the same triggering event hold the same parameters in their argument lists.
  - The list of parameters associated with events does not contain any redundancies.

4.3.2 The Original Design

In Figure 21, we show the original class diagram of the translator. The old class diagram of the translator is the model of the data structure, where each abstract data type is represented as a class. It is divided into three packages: a class diagram for the internal structures of the GRC specifications, another one for the internal structures of the SCS specifications, and one for the message sequence entries. Here we are only interested in the class diagram representing the GRC structures since it is the only one affected by the changes needed for the parameterized events. The main class in this diagram is the GRC class. It is an aggregation of all the other classes that represent the components of a GRC as described in Chapter 2.

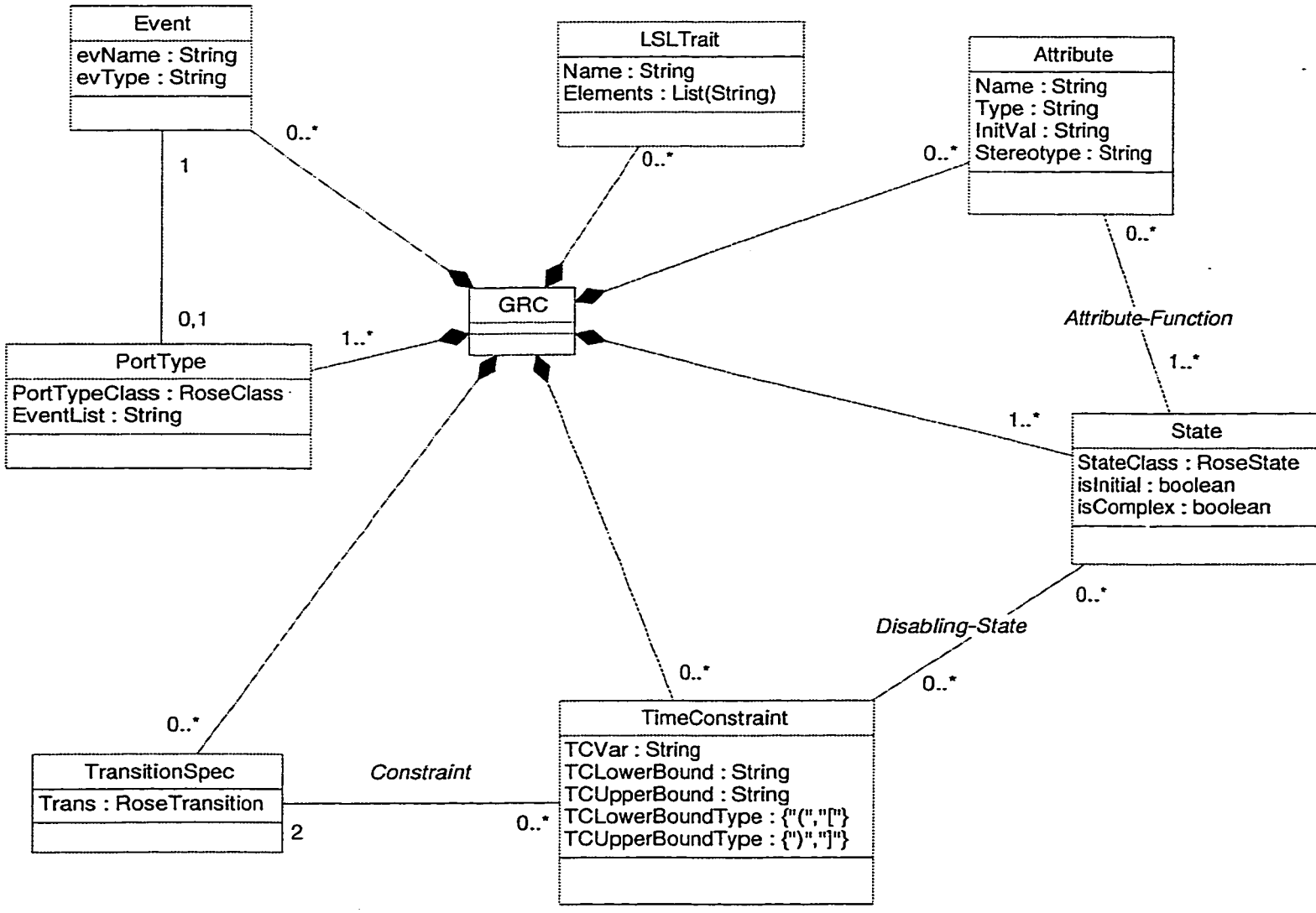

Figure 21: Rose-GRC Translator Class Diagram (old)

4.3.3 The Modified GRC-Translator

In order to manage parameters in the translator, we create a new class, *Parameter*. This class is an aggregate of the class GRC. It is associated with class *Event* with the *Arguments* association, and class *State* with the association *Parameter-Function*. It is also an inheritance of the class *Attribute*. The new class diagram is illustrated in Figure 22.

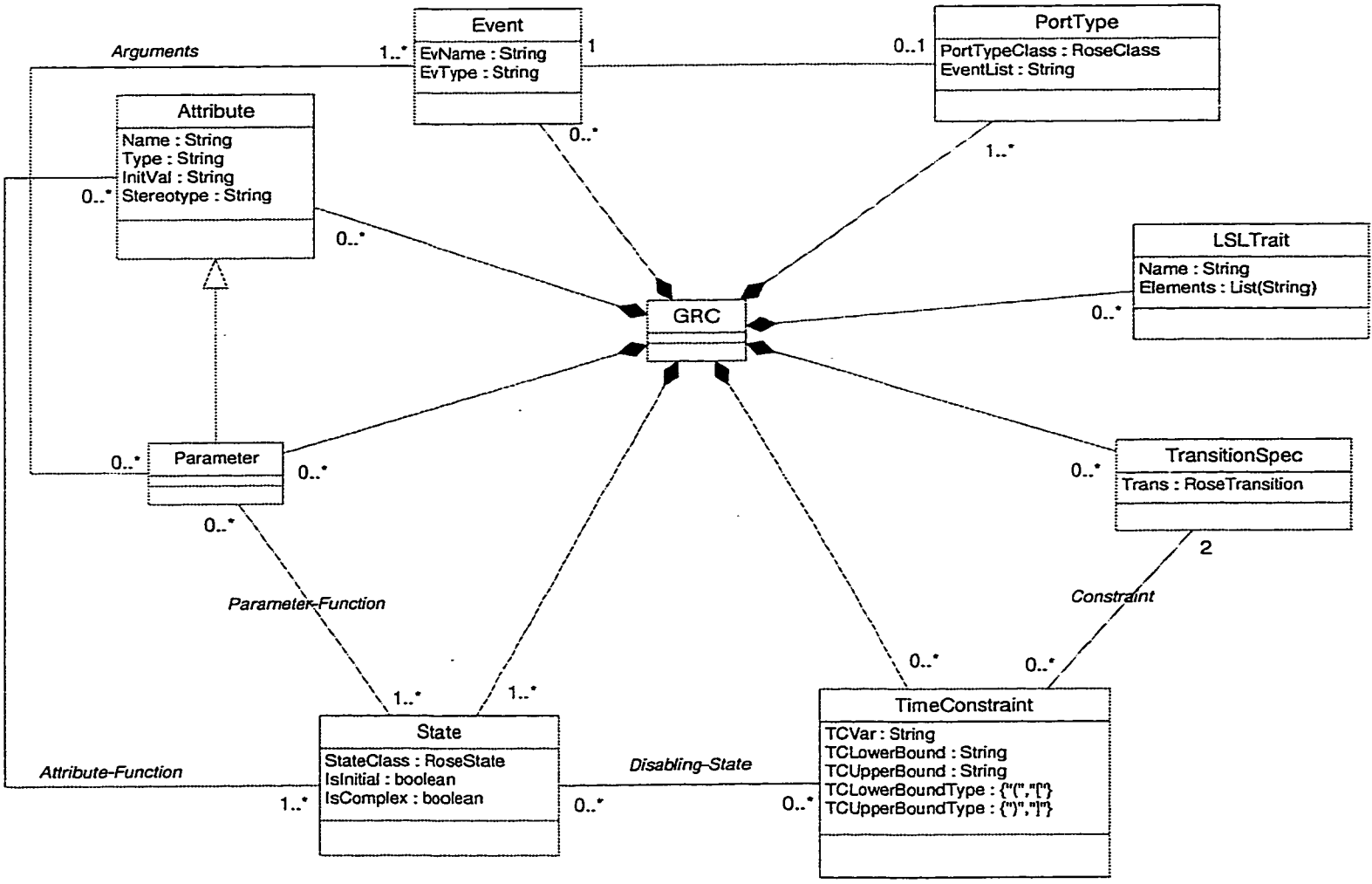

Figure 22: Rose-GRC Translator Class Diagram (new)

4.4 Interpreter

In this section, we describe how the existing interpreter is improved to guarantee correct acquisition of the event parameterization in the TROM class. In the following, we describe the requirements of the Interpreter so that it conforms with the formal representation of parameters, the original design of the Interpreter, and the new design that handles parameterized events. The algorithms of the corresponding implementations are presented in Appendix A.

4.4.1 Upgrading the Interpreter

Here, we discuss the required changes to the interpreter.

- *Abstract Syntax Tree structure:* The AST structure should be modified to include the representation of parameters. After parsing the TROM files, the AST is generated by the interpreter. Therefore, new data structures are added to the interpreter for parameters. The two main additions to the AST are:
  - The original AST includes a list of events. The new AST should include in each event a list of parameters associated with the event. The parameter list represents all the possible parameters that can be carried by the event's argument list.
  - The original AST has a list of transition specifications. The new AST should include within each transition specification a list of parameter entries, where each entry represents a parameter and its value.
- TROM *class parser:* The TROM class parser is also modified to parse the parameters. The following changes and additions should be made:
  - a new parser is added to parse the parameter specifications. It performs a syntax analysis on the Parameter-Specifications section according to the TROM grammar. Then, to each event in this section, it assigns the parameters to the event's parameter list in the data structure.
  - The parser of the Transition-Specifications section is upgraded to include

the functionality of parsing the argument list of the triggering event. After performing a syntax check according to the grammar, the parser extracts the parameter entries and assigns them to the parameter list of the transition data structure (a data structure that represents Transition-Specifications). The entry could be a parameter name, or a parameter followed by a value.

- *Semantic checks:* while parsing the parameters, the interpreter performs some sort of semantic analysis to verify the compliance of the representation to certain rules. In case of violation, appropriate error messages are generated. The parameter representation should respect the following rules:
  - In the Parameter-Specifications section, each event should be valid in the event list (the data structure that holds all the defined events of the class).
  - In the Parameter-Specifications section, each parameter should be valid in the attribute list (the data structure that holds all the defined attributes and parameters of the class).
  - In the argument list of the Transition-Specifications section, each parameter should be valid in the attribute list. In addition, each parameter should be valid in the parameter list of the triggering event.

4.4.2 Interpreter Design Without Parameters

In Figure 23 and Figure 24, we show the high level class diagrams of the original *Interpreter*. A detailed class diagram is shown in Figure 25 and Figure 26. An abstract syntax tree is an aggregation of *LSL trait, TROMclass, SCS, and SCSSimEv*. These are the classes that model the entities in the three tiers, and the simulation events. The detailed class diagram for each class shows the internal structures and the interface. Modularity in the design and coupling between classes are explicit in the diagrams. Any change in any one of the classes will not affect the others.

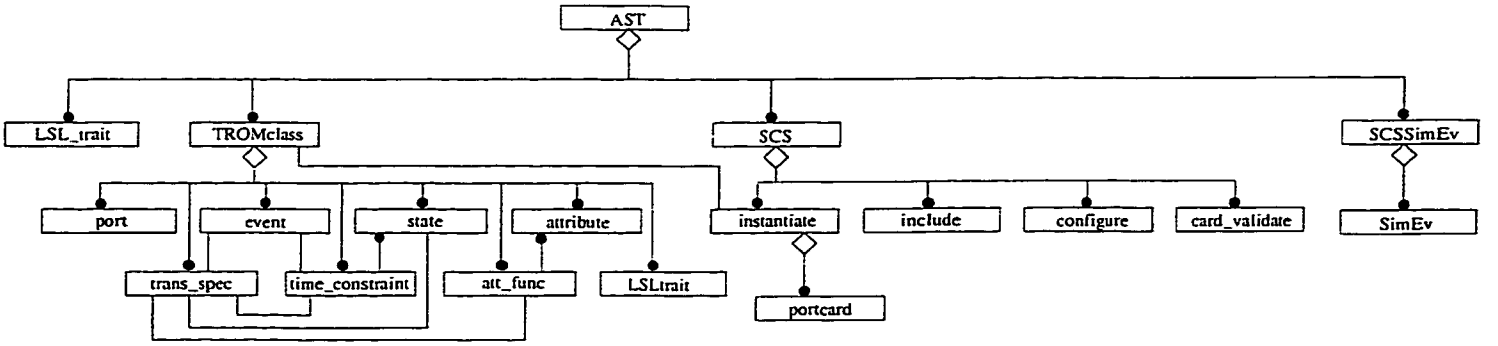

Figure 23: Interpreter Class Diagram (old)

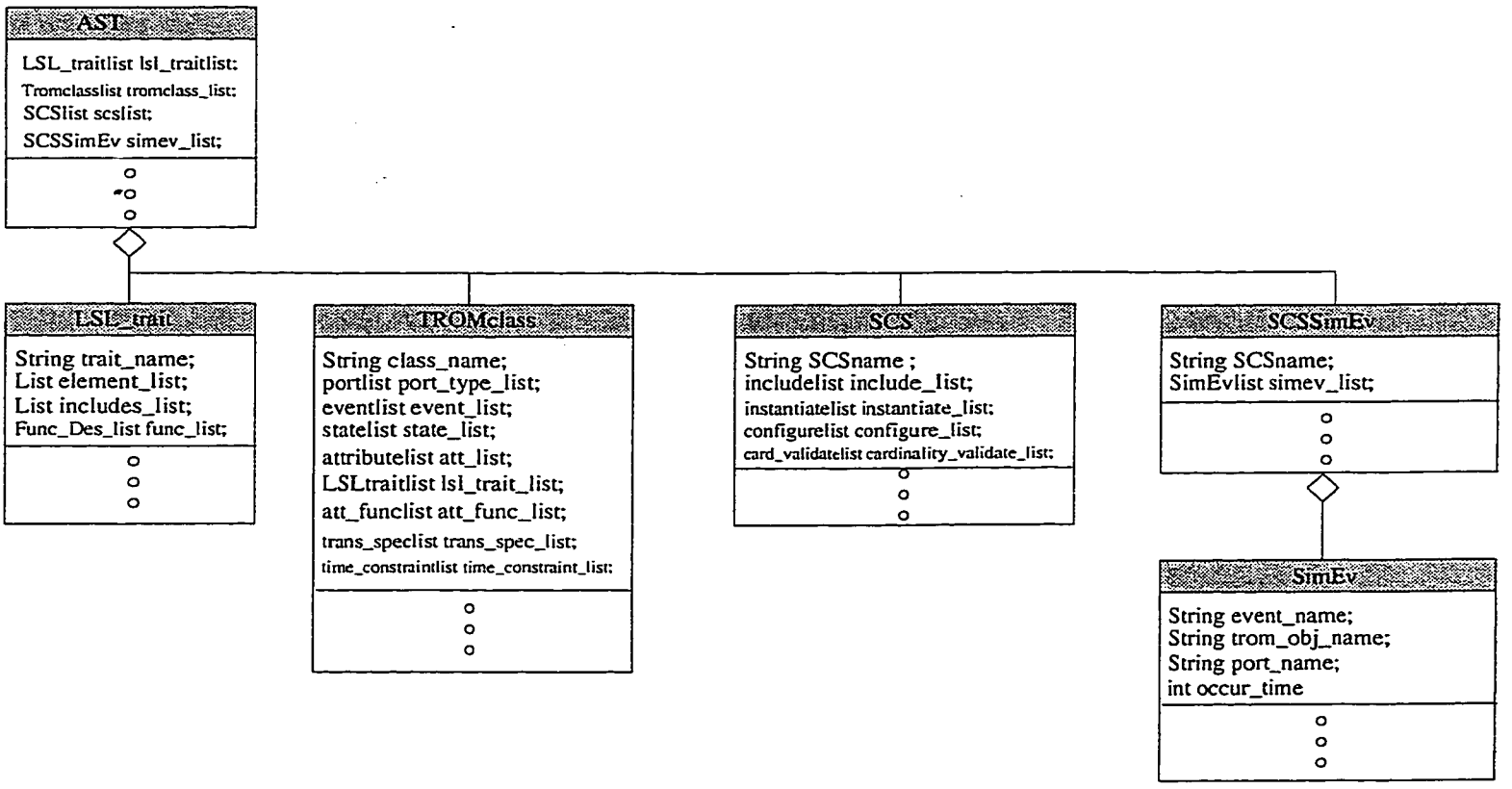

Figure 24: Interpreter Class Diagram - Detailed (old)

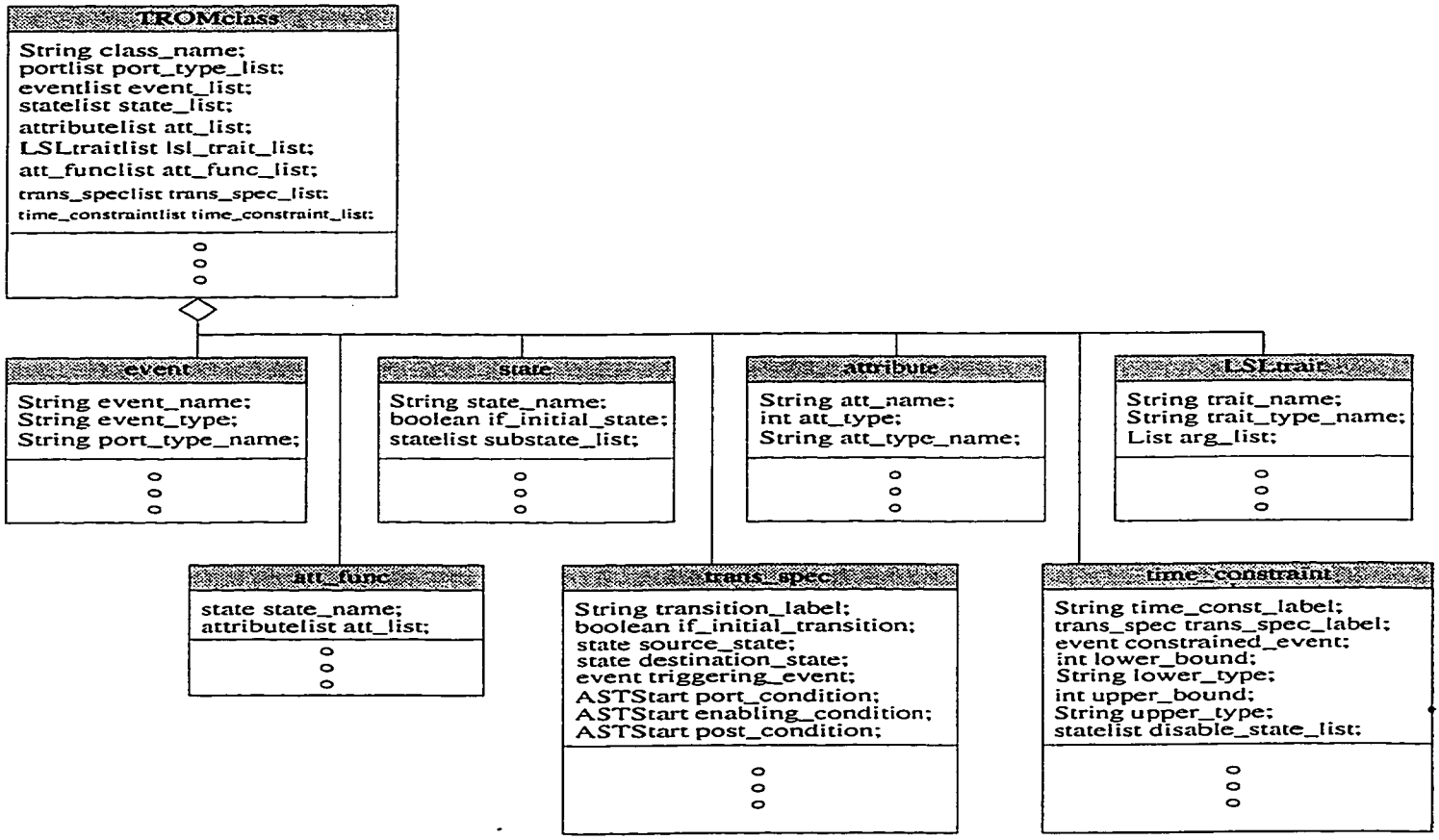

Figure 25: Interpreter Class Diagram - *TROMclass(old)*

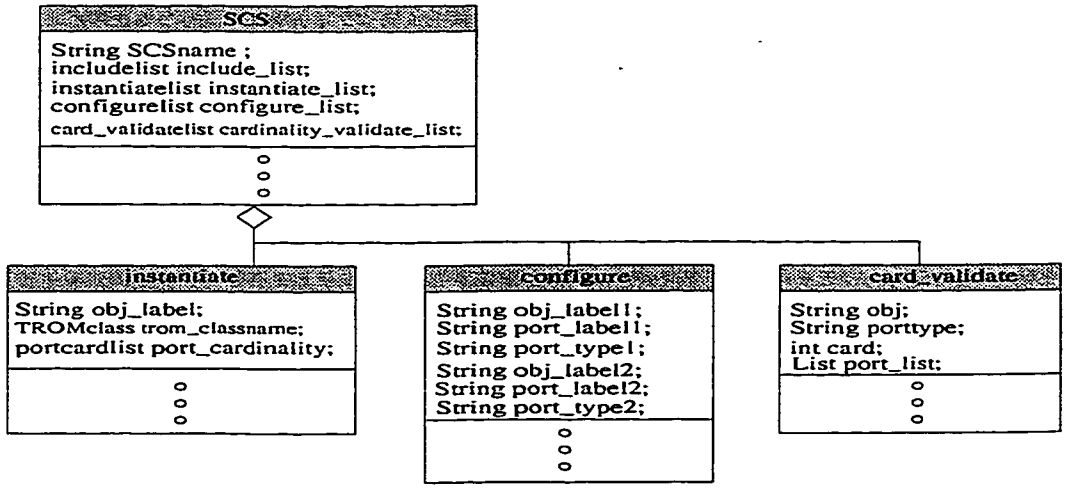

Figure 26: Interpreter Class Diagram - SCS (old)

4.4.3 Interpreter with Parameters

To accommodate parameterized events in the interpreter, the only class diagram that needs to be modified is the *TROMclass* class diagram. Therefore, we present in Figure 27 the modified high level class diagram, and consequently, the modified *TROMclass* class diagram in Figure 28. The latter shows two added classes *Parameter* and *Trans_parameter*. *Parameter* class is an aggregation of the class *Event*. *Trans_parameter* class is an aggregation of the class *trans_spec*, it is also an aggregate of the class *Parameter* since it represents a parameter and it's corresponding value. The data structure of parameters is fully described in Appendix A.

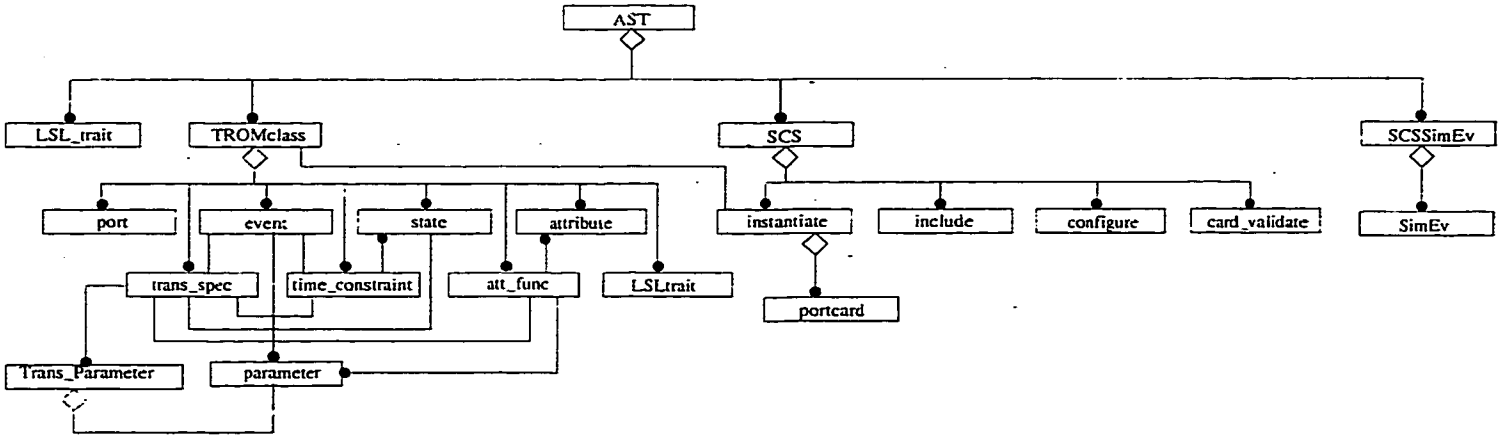

Figure 27: Interpreter Class Diagram (new)

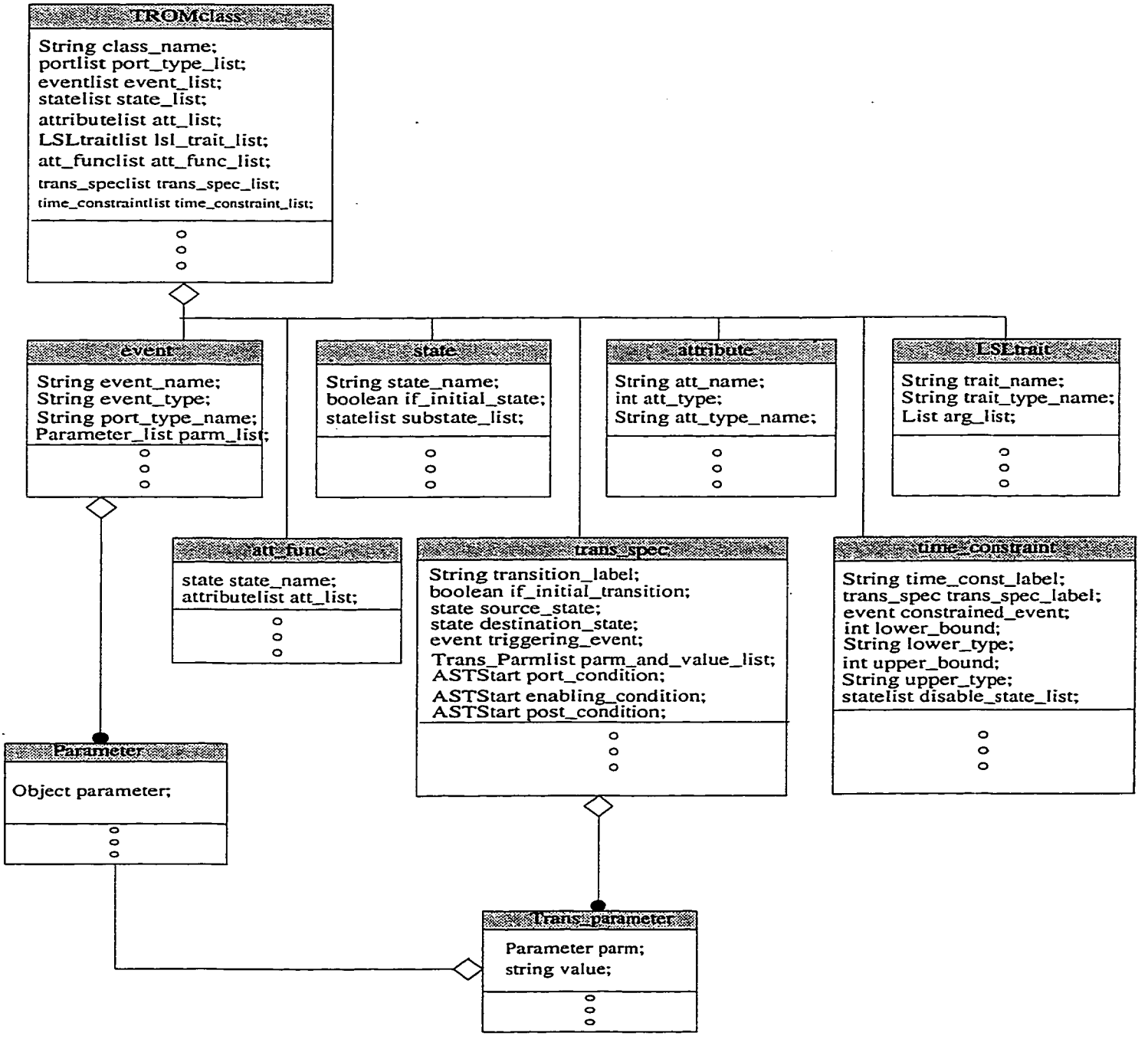

Figure 28: Interpreter Class Diagram - TROMclass (new)

4.5 Remodeling the Train-Gate-Controller System

Train-Gate-Controller (TGC) problem is a bench-mark case study adopted by the real-time research community to illustrate specification and verification methods. In this section we show the models of TGC problem without parameters. These models have been used repeatedly by Achuthan [Ach95] and Muthiayen [Mut96]. We also show the models with parameterized events and comment on the clarity of the new models.

In the TGC system, several trains cross a gate independently and simultaneously using nonoverlapping tracks. A train may choose to cross any gate on its way. A controller controls each gate. When a train approaches the gate, it sends a message to the associated controller. The controller commands the gate to close. When the train exits the crossing, it sends a message to the controller, which instructs the gate to open.

To ensure safety of this system, the following timing constraints are placed on messages. A train should be inside the crossing 2 to 4 time units after sending the message indicating that it is approaching the gate. The train should send the message indicating that it is ready to exit the crossing within 6 time units from the first message. Within 1 time unit from receiving the initial message from the train, the controller must instruct the gate to lower and the controller starts monitoring the gate. If the controller receives an approaching message while it is monitoring the gate, it should continue monitoring the gate. The controller will instruct the gate to raise within 1 time unit after the last train exits the crossing. The gate must close within 1 time unit after the controller instructs it to lower. The gate must open within 1 time unit after the controller instructs it to raise.

4.5.1 Original Models

Here, we present the original model of the Train-Gate-Controller system designed in Rational Rose and the formal specifications generated by the Rose-GEC Translator.

The class diagram presented in Figure 29 shows the three GRC classes: *Train*, *Gate* and *Controller*.
Train GRC is an aggregate of port types *@C*.
Controller GRC is an aggregate of port types *@G* and *@P*.
Gate GRC is an aggregate of port types *@S*.
An association exists between port type *@C* of *Train* and *@P* of *Controller*, meaning that the generic class *Train* uses port type *@C* to communicate to the generic class *Controller* through its port type *@P*.
An association exists between port type *@S* of *Gate* and port type *@G* of *Controller*, meaning that generic class *Controller* uses port type *@G* to communicate with generic class *Gate* through its port type *@S*.
Train GRC has one port type *@C* where the following events may occur: output event *Near*, output event *Exit*.
Train GRC has one attribute, *cr* of type *@C*, which is the port type of the *Train* class.
Controller GRC has two port types: *@P* and *@G*. At port type *@P*, the following events may occur: input event *Near*, input event *Exit*. At port type *@G*, the following events may occur: output event *Lower*, output event *Raise*.
Controller GRC has one data type attribute, *inSet*. The type is an abstract data type defined in the LSL trait *Set* with parameters *@P* and *PSet*, where *@P* is the type of each name and *PSet* is the name of the abstract data type.
Gate GRC has one port type*@S* where the following events may occur: input event *Lower*, input event *Raise*.

Train GRC

The statechart diagram for *Train* in shown in Figure 30. A *Train* object can be in one of four states: *idle*, *toCross*, *cross*, *leave* with *Idle* as the initial state.

When event *Near* occurs in state *idle*,attribute *cr* is set to *pid*, the identifier of the port where *Near* occurs. This transition is the constraining transition for two time

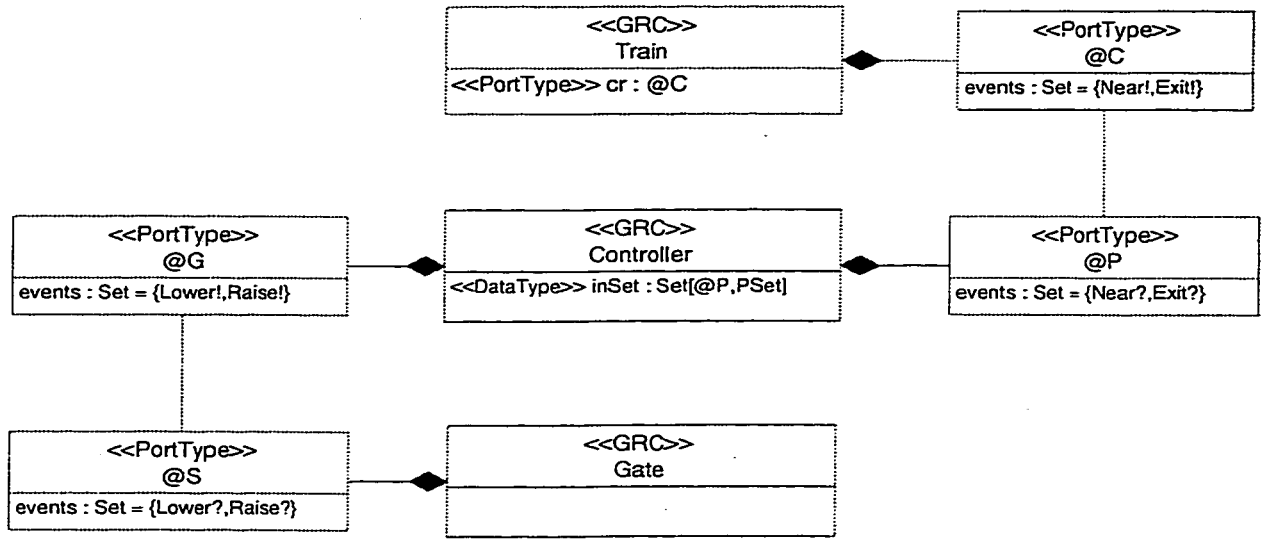

Figure 29: Train-Gate-Controller class diagram

constraints, labeled *TCvar1* and *TCvar2*. *Train* goes into state *toCross*.
A transition from state *toCross* to *cross* is triggered when internal event *In* occurs in state *toCross*, and if the time constraint condition $TCvar1 \geq 2$ AND $TCvar1 \leq 4$ is true. This time constraint means that internal event *In* should occur within 2 to 4 time units after event *Near* occurs in state *idle*.
When internal event *Out* occurs in state *cross*, *Train* goes into state *leave*.
A transition from state *leave* to *idle* is triggered when event *Exit* occurs in state *leave*, and if the time constraint condition $Tcvar2 \leq 6$ is true. This time constraint means that event *Exit* should occur within 6 time units after event *Near* occurs in state *idle*.

Controller GRC

The statechart diagram for *Controller* is shown in Figure 31. A *Controller* object can be in one of four states: *idle*, *activate*, *monitor*, *deactivate* with *Idle* as the initial state.

When event *Near* occurs in state *idle*, the attribute *inSet* is modified to include the new entry *pid* (*pid* is the identifier of the port where *Near* occurs). The *Controller* goes into state *activate*. This transition is the constraining transition for time constraint *TCvar1*.
When event *Near* occurs in state *activate* from a different *Train* (*pid* is not already a member of set *inSet*), the attribute *inSet* is modified to include the new *pid* (identifier

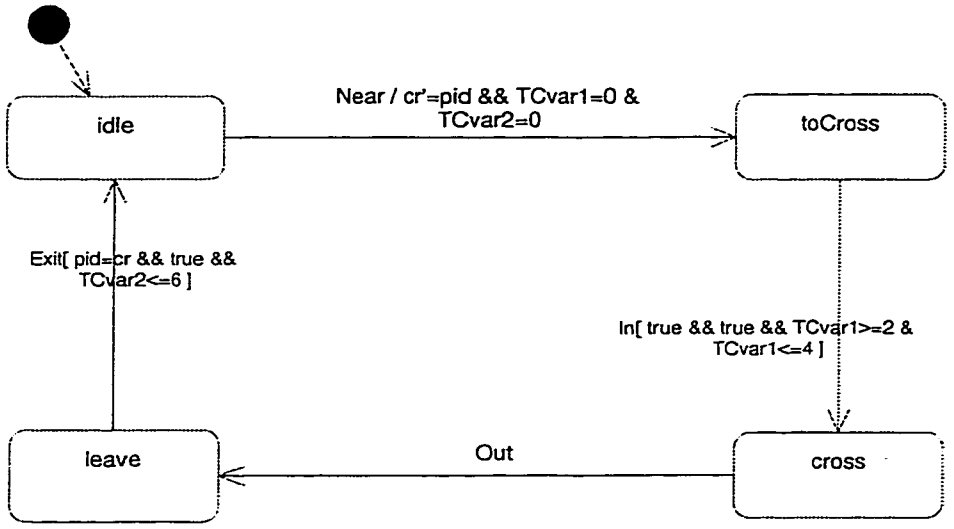

Figure 30: Train StateChart diagram

of the port where the new event *Near* occurs). The *controller* remains in the state *activate*.

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

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

When event *Exit* occurs in state *monitor*, if the identifier (*TID*) of the train which event *Exit* was received is a member of *inSet* and if the size of *inSet* is greater than 1, meaning that more than one *Trains* are in the crossing, then the current *TID* is deleted from *inSet* and *Controller* remains in state *monitor*.

When event *Exit* occurs in state *monitor*, if the identifier (*TID*) of the train which event *Exit* was received is a member of *inSet* and if the size of *inSet* is equal to 1, meaning that this is the only *Train* in the crossing, then the current *pid* is deleted from *inSet* and *Controller* goes into state *deactivate*. This is the constraining transition for time constraint *TCvar2*.

When event *Raise* occurs in state *deactivate*, if time constraint condition $Tcvar2 \leq 1$ is true, the *Controller* goes into state *idle*. This time constraint condition means that

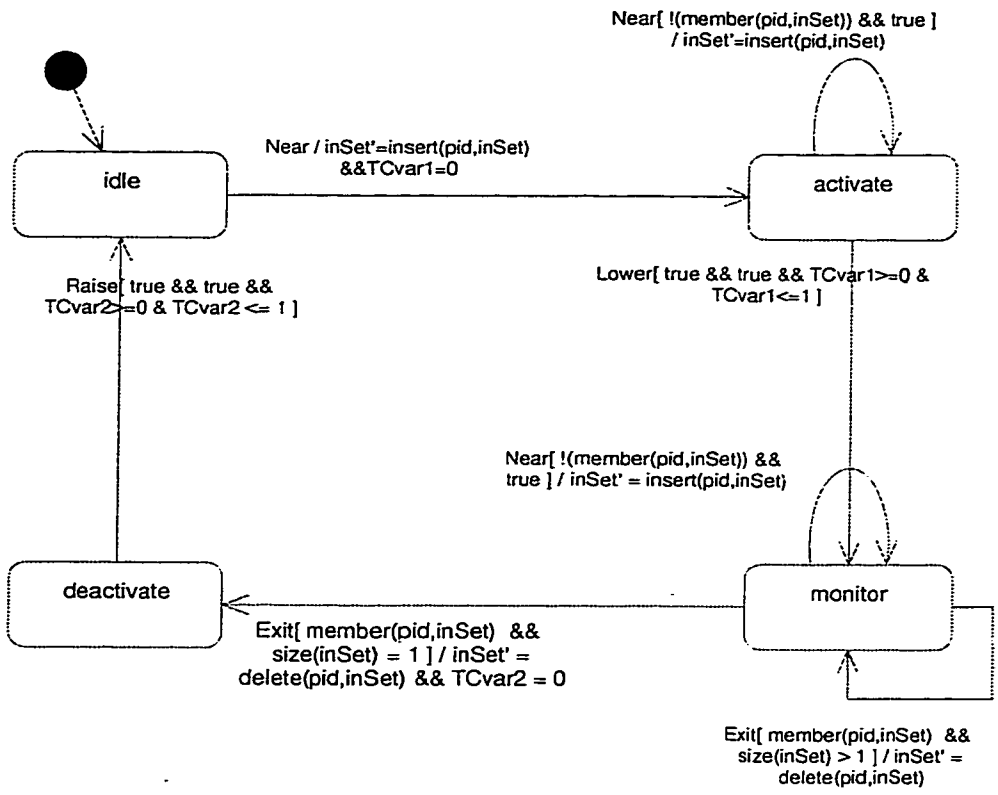

Figure 31: Controller StateChart diagram

event *Raise* should occur within 1 time unit after event *Exit* was received from the last *Train* in the crossing.

Gate GRC

The statechart diagram for *Gate* in shown in Figure 32. A *Gate* object can be in one of four states: *opened*, *toClose*, *closed*, *toOpen* where *Closed* is the initial state.

When event *Lower* occurs in state *opened*, the *Gate* goes into state *toClose*. This is the constraining transition for time constraint labeled *Tcvar1*.
A transition from state *toClose* to *closed* is triggered when internal event *Down* occurs in state *toClose* if the time constraint condition $TCvar1 \leq 1$ is true. This time constraint means that internal event *Down* should occur within 1 time unit after event *Lower* occurs in state *opened*.
When event *Raise* occurs in state *closed*, the *Gate* goes into state *toOpen*. This is the constraining transition for time constraint *TCvar2*.

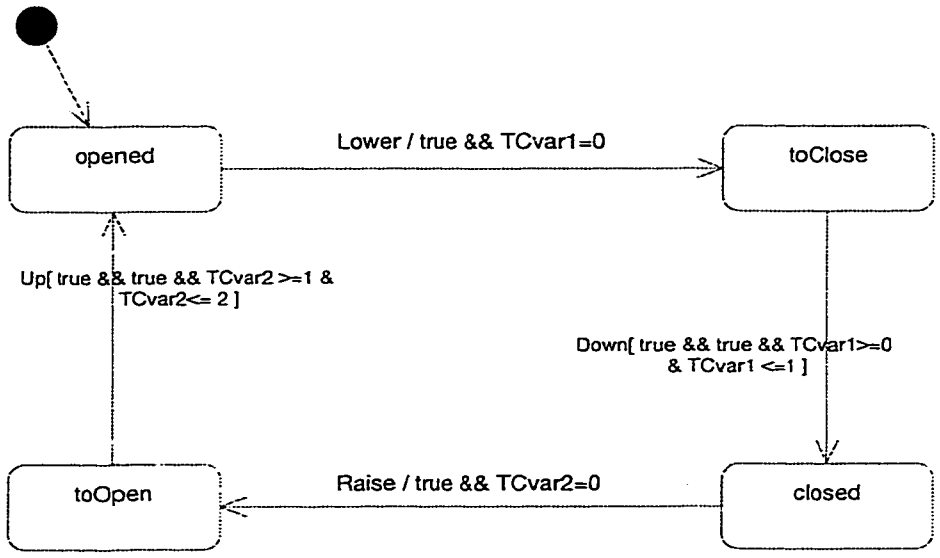

Figure 32: Gate StateChart diagram

A transition from state *toOpen* to *open* happens when internal event *Up* occurs in state *toOpen* if the time constraint condition $TCvar2 \geq 1$ and $Tcvar2 \leq 2$ is true. This time constraint means that internal event *Up* should occur within 1 to 2 time units after event *Raise* occurs in state *closed.*

Formal Specifications

Figures 33, 34, 35 show the original formal specifications of the TGC system.

4.5.2 Revised Models

Here, we present the revised model of the Train-Gate-Controller system with parameterized events. In this model, we show the expressiveness and clarity provided by the parameterization of events to the model. This is achieved by adding a train ID to the system. Each time a train needs to cross a gate, it is identified by its ID passed as a parameter with the *Train* incoming events. This ID replaces the usage of the pid at which the *Train* and *Controller* communicate with each other.

Figure 37 shows the modified class diagram for the TGC system. The class diagram structure is the same as the one of the original model except for few modifications: Class *Train* has a parameter *TID* of type integer representing the train ID crossing the gate.

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

Figure 33: Train class specifications

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

Figure 34: Formal specification for GRC Gate

```
Class Controller [@P, @G]
Events: Lower!@G, Near?@P, Raise!@G, Exit?@P
States: *idle, activate, deactivate, monitor
Attributes: inSet:PSet
Traits: Set[@P,PSet]
Attribute-Function: activate → {inSet}; deactivate → {inSet}; monitor → {inSet};
          idle → {};
Transition-Specifications:
     R1: <activate,monitor>; Lower(true);
          true ⟹ true;
     R2: <activate,activate>; Near(!(member(pid,inSet)));
          true ⟹ inSet/=insert(pid,inSet);
     R3: <deactivate,idle>; Raise(true);
          true ⟹ true;
     R4: <monitor,deactivate>; Exit(member(pid,inSet));
          size(inSet)=1 ⟹ inSet/=delete(pid,inSet);
     R5: <monitor,monitor>; Exit(member(pid,inSet));
          size(inSet)>1 ⟹ inSet/=delete(pid,inSet);
     R6: <monitor,monitor>; Near(!(member(pid,inSet)));
          true ⟹ inSet/=insert(pid,inSet);
     R7: <idle,activate>; Near(true);
          true ⟹ inSet/=insert(pid,inSet);
Time-Constraints:
     TCvar1: R7, Lower, [0, 1], {};
     TCvar2: R4, Raise, [0, 1], {};
end
```

Figure 35: Controller class specifications

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

Figure 36: SCS for Train-Gate-Controller

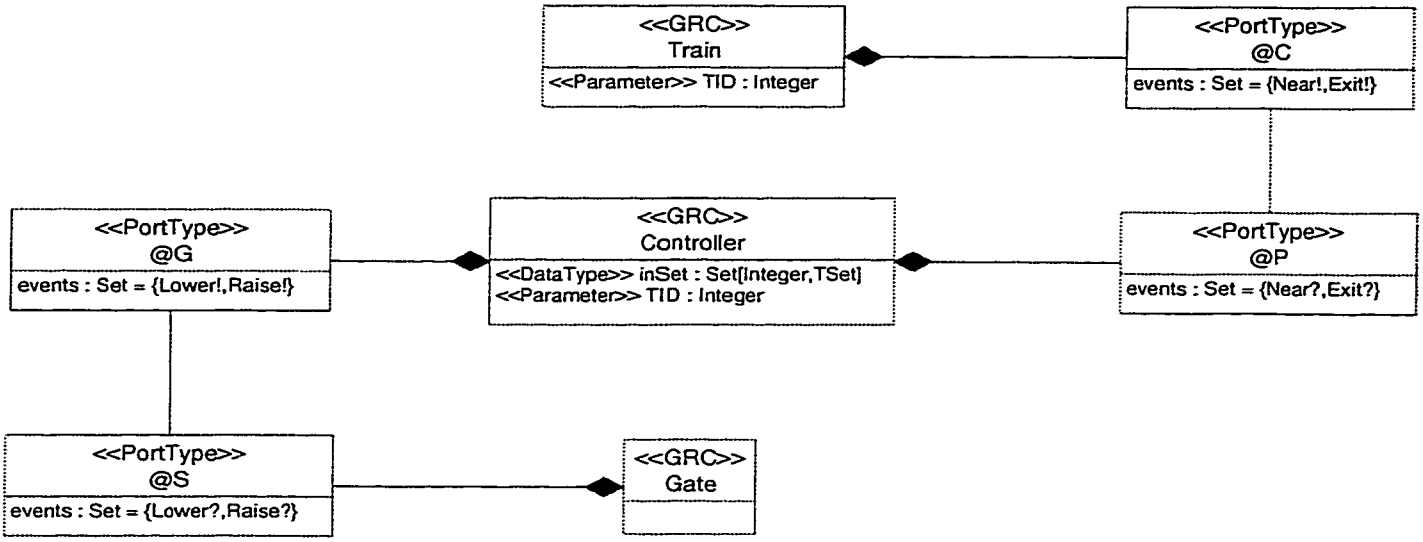

Figure 37: Train-Gate-Controller class diagram with parameters

Controller GRC has one data type attribute, *inSet*. The type is an abstract data type defined in the LSL trait *Set* with parameters *Integer* and *TSet*, where *Integer* is the type of each name and *TSet* is the name of the abstract data type. It also has a parameter *TID* of type integer.

In the following, we show the new statecharts of the *Train* and *Controller* classes. The *Gate* class statechart diagram is not modified so we do not present it.

Train GRC

The modified statechart diagram for *Train* in shown in Figure 38. The behavior remains the same after remodeling the TGC system. Few changes has been applied to the transitions to accomodate parameters. When event *Near* occurs in state *idle*, the parameter *TID* is passed with the event to indicate which train needs to cross. In this case, there is no need to store the pid where the *Train* communicates with the *Controller*.

Controller GRC

The new statechart diagram for *Controller* is shown in Figure 39. It adopts the same behavior as the original statechart of the *Controller* class. However, few changes have been applied to accommodate the parameterization of events. When event *Near* occurs in state *idle*, the attribute *inSet* is modified to include the new entry *TID*

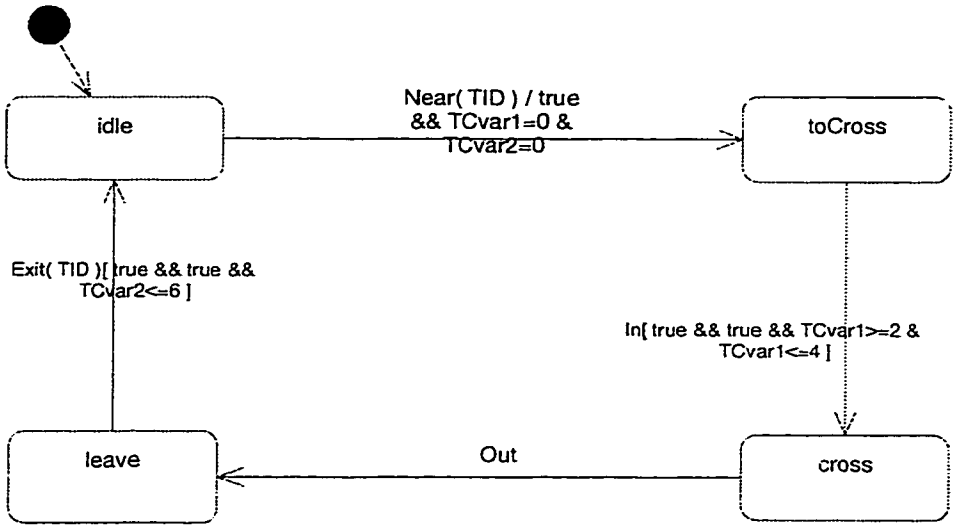

Figure 38: Train StateChart diagram with parameterized events

(*TID* is the train identifier which sends the message *Near*). The *Controller* goes into state *activate*. In this case, there is no need to store the pid of the port at which the *Controller* communicates with the *Train*. When event *Near* occurs in state *activate* from a different *Train* (its *TID* is not already a member of set *inSet*), the attribute *inSet* is modified to include the new *TID* (train identifier which sends the new event *Near*). The *controller* remains in the state *activate*. When event *Near* occurs in state *monitor* from a different *Train* (its *TID* is not already a member of set *inSet*), the attribute *inSet* is modified to include the new *TID* (train identifier which sends the new event *Near*). The *Controller* remains in state *monitor*.

When event *Exit* occurs in state *monitor*, if the identifier (*TID*) of the train which event *Exit* was received is a member of *inSet* and if the size of *inSet* is greater than 1, meaning that more than one *Trains* are in the crossing, then the current *TID* is deleted from *inSet* and *Controller* remains in state *monitor*.

When event *Exit* occurs in state *monitor*, if the identifier (*TID*) of the train which event *Exit* was received is a member of *inSet* and if the size of *inSet* is equal to 1, meaning that this is the only *Train* in the crossing, then the current *TID* is deleted from *inSet* and *Controller* goes into state *deactivate*. This is the constraining transition for time constraint *TCvar2*.

Formal Specifications

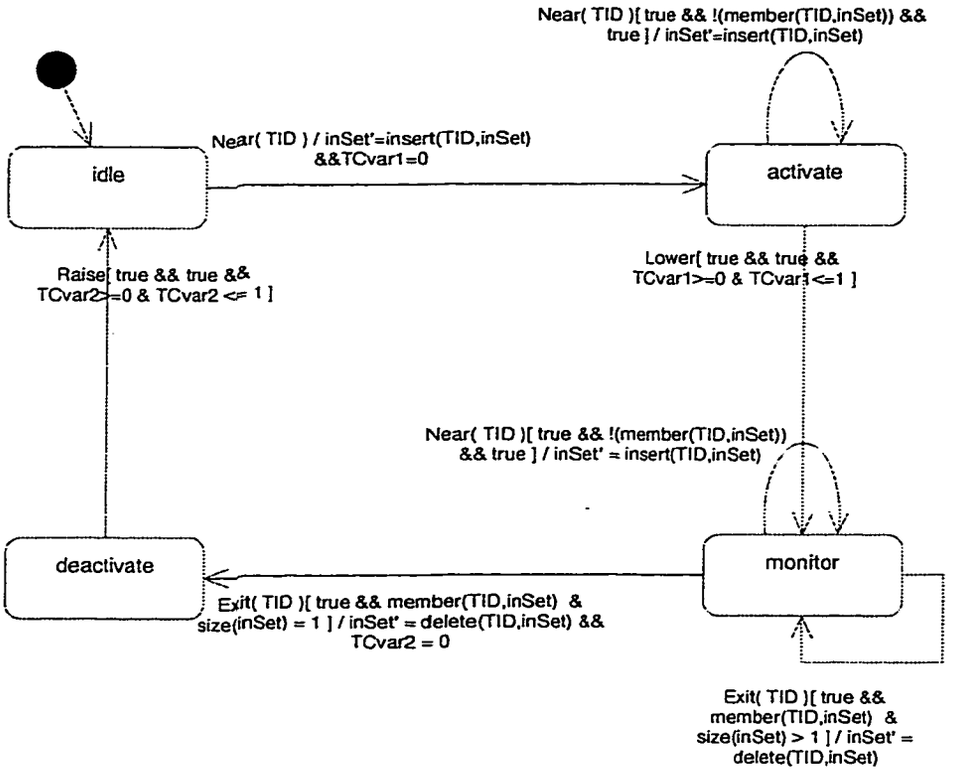

Figure 39: Controller StateChart diagram with parameterized events

In Figures 40, 41, 42 we show the formal specificaitions of the classes Train, Gate, and Controller generated by the new version of the Translator.

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

Figure 40: Formal specification for GRC Train with parameters

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

Figure 41: Formal specification for GRC Gate with parameters

```
Class Controller [@P, @G]
Events: Lower!@G, Near?@P, Raise!@G, Exit?@P
States: *idle, activate, deactivate, monitor
Attributes: inSet:TSet;TID:Integer
Traits: Set[@P,PSet]
Attribute-Function: activate → {inSet, TID}; deactivate → {inSet}; monitor → {inSet, TID};
        idle → {TID};
Parameter-Specifications:
    Exit: TID;
    Near: TID;
Transition-Specifications:
    R1: <activate,monitor>; Lower[ ](true);
        true ⟹ true;
    R2: <activate,activate>; Near[TID](true); !(member(TID,inSet))
        ⟹ inSet/=insert(TID,inSet);
    R3: <deactivate,idle>; Raise[ ](true);
        true ⟹ true;
    R4: <monitor,deactivate>; Exit[TID](true); member(TID,inSet) &
        size(inSet)=1 ⟹ inSet/=delete(TID,inSet);
    R5: <monitor,monitor>; Exit[TID](true);member(TID,inSet) &
        size(inSet)>1 ⟹ inSet/=delete(TID,inSet);
    R6: <monitor,monitor>; Near[TID](true);!(member(TID,inSet))
        ⟹ inSet/=insert(TID,inSet);
    R7: <idle,activate>; Near[TID](true);
        true ⟹ inSet/=insert(TID,inSet);
Time-Constraints:
    TCvar1: R7, Lower, [0, 1], {};
    TCvar2: R4, Raise, [0, 1], {};
end
```

Figure 42: Formal specification for GRC Controller with parameters

Chapter 5

Asynchronous Transfer Mode - Available Bit Rate Protocol

In this chapter, we apply our approach to parameterize the events in the TROM formalism to modeling the Available Bit Rate (ABR) Protocol of the Asynchronous Transfer Mode (ATM). We present an overview of the ATM and the ABR protocol in Section 6.1. We state the problem of the ABR model in section 6.2. In section 6.3 we describe the detailed behavior of the ABR protocol. Then, in section 6.4, we model the ABR protocol using Real-Time UML notation. In section 6.5, we produce the complete formal specifications of the protocol using the extended Rose-GRC Translator.

5.1 ATM ABR Protocol

Asynchronous Transfer Mode (ATM) is a multiplexing scheme and switching technique tailored for high speed networks [HRS98]. It addresses the needs of carriers to efficiently move all types of information from one place to another across a shared transport medium. The efficiencies of ATM depend on two characteristics: fixed-length cells and logical networking. ATM technology provides a wide variety of services and applications of networking. The control of ATM network traffic is a very crucial issue. Therefore, it is achieved via the ATM Traffic Management protocols. The main role of Traffic Management is to protect the network from congestion and promote an efficient use of network resources. It specifies five classes of protocols:

constant bit rate (CBR), real-time variable bit rate (rt-VBR), non-real time variable bit rate (nrt-VBR), unspecified bit rate (UBR), and available bit rate (ABR) [ATM96]. In this thesis, we consider the Available Bit Rate (ABR) protocol for which the English specification is stated in the Traffic Management Specification Version 4.0 of the ATM forum.

5.1.1 ABR

The ABR service uses a mechanism based on congestion control and avoidance that depend on the rate of cells transmitted across the network [JR96]. It deals with bi-directionally-connected end-systems: sending sources and receiving destinations. Each end-system is both a source and a destination. The network consists of switches which calculate the allowable rate for the sources. This rate is sent to the sources as feedback via resource management (RM) cells. These cells are periodically sent by the source after transmitting a certain number of data cells (every *Nrm* data cells). When the destination receives an RM cell, it turns it back to the source traversing through switches.
In order to maintain efficient communication on the network, switches can inform the corresponding sources of the state of the network using one of three options [JR96]:

- A congested switch can set a bit (the Explicit Forward Congestion Indication (EFCI)) in the header of each data cell.
- A congested switch can set two bits (Congestion Indication (CI) bit and No Increase (NI) bit)in the RM cells.
- Congested switches can reduce in the RM cells the value of ER, Explicit Rate, field to any desired value.

Usually RM cells are generated by the source. Switches on the network update the corresponding fields in the received RM cells based on congestion information available. However, under extreme congestion, a switch can optionally generate a RM cell and send them immediately to the source. This mechanism is called backward explicit congestion notification (BECN) [JR96].

5.1.2 ABR Parameters

For a connection to be established, some parameters need to be set to guarantee proper communication between the entities [JR96]. The peak cell rate (PCR) parameter defines the maximum rate at which a source is allowed to transmit. There is also the minimum cell rate (MCR) parameter, which indicates the minimum rate at which a source can transmit. Normally, the network guarantees a transmission bandwidth between PCR and MCR. The allowed cell rate (ACR), on the other hand, expresses the rate at which a source is allowed to send cells at any time. The value of ACR varies between MCR and PCR based on the congestion information. Initially, ACR is initialized to a fixed value expressed by the initial cell rate (ICR). Table 12 shows the list of parameters used in the ABR mechanism [ATM96]. These parameters are system parameters. They are not associated with cells and their values are constant except for ACR. The rest of the parameters are explained as they occur in this chapter.

5.1.3 Forward and Backward RM cells

Resource Management cells travel from the source to the destination that turns them back to the source on the same path. As a convention, the direction from source to destination is called the forward direction and the one from destination to source is called the backward direction. Therefore, RM cells travelling from source to destination are called Forward RM (FRM) cells, and The ones travelling from the destination to the source are called Backward RM (BRM) cells [JR96].
When there is bi-directional traffic, sources are at the same time destinations and destinations are at the same time sources, and FRMs and BRMs travel in both directions. To distinguish FRMs from BRMs, a bit in the RM cell is used to indicate its direction. This direction bit (DIR) is changed from 0 to 1 by the receiving destination to indicate that the cell has been changed from FRM to BRM [JR96]. Figure 43 illustrates FRM and BRM cells.

We should mention here that there are two kinds of RM cells. In-rate RM cells and out-of-rate RM cells. The in-rate RM cells are the ones generated by the source and consist part of its network load, i.e., the RM cells and data cells sent by the source should not exceed ACR.

PCR		The Peak Cell Rate is the cell rate which the source may never exceed.
MCR		The Minimum Cell Rate is the rate at which the source is always allowed to send.
ACR		The Allowed Cell Rate is the current rate at which a source is allowed to send.
ICR		The Initial Cell Rate is the rate at which a source should send initially and after an idle period.
Nrm		is the maximum number of cells a source may send for each forward RM-cell.
Mrm		controls allocation of bandwidth between forward RM-cells, backward RM-cells, and data cells.
Trm		provides an upper bound on the time between forward RM-cells for an active source.
RIF		Rate Increase Factor controls the amount by which the cell transmission rate may increase upon receipt of an RM-cell.
RDF		The Rate Decrease Factor, RDF, controls the decrease in the cell transmission rate.
ADTF		The ACR Decrease Time Factor is the time permitted between sending RM-cells before the rate is decreased to ICR.
CRM		Missing RM-cell count. CRM limits the number of forward RM-cells which may be sent in the absence of received backward RM-cells.
CDF		The Cutoff Decrease Factor, CDF, controls the decrease in ACR associated with CRM.

Table 12: List of ABR Parameters

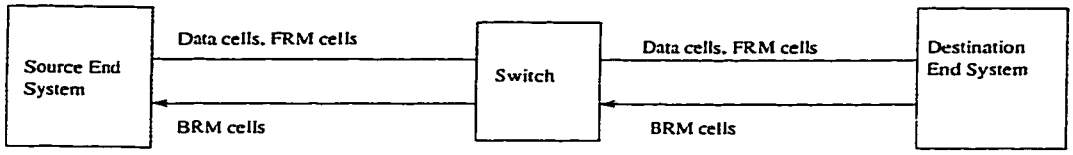

Figure 43: Flow of data,FRM,and BRM cells

However, the out-of-rate RM cells can be generated by the switch, destination, or even the source under exceptional circumstances. The out-of-rate RM cells are not counted in the ACR of the source. The network can carry them if enough bandwidth is available or they can be discarded if the network is congested. A switch can generate an out-of-rate RM cell for BECN. Also, sources can detect the network by sending an out-of-rate RM cell only if its ACR is set to zero by the network. On the other hand, destinations may generate an out-of-rate RM cell if the source's ACR is zero or is not enough to return all the RM cells the destination receives [JR96].

RM cell Format

Figure 44 shows the format of the RM cell. It consists of several fields; here we are only interested with some of these fields and they are:

- The direction (DIR) bit is to distinguish between forward and backward RM cells.
- The backward notification (BN) bit is set by switch generated BECN cells.
- The congestion indication (CI) bit is set by congested switches.
- The no increase (NI) bit is used to indicate moderate congestion, it means that no increase of the rate is necessary.
- The current cell rate (CCR) field is set by the source when it sends the RM cell to indicate its current rate to the network.
- The minimum cell rate (MCR) field, like PCR, ICR, and other parameters does not change as long as the connection is still set.
- The explicit rate (ER) field, like CI and NI, is used by the network to provide feedback to the source. The ER field indicates the maximum rate allowed to the source.

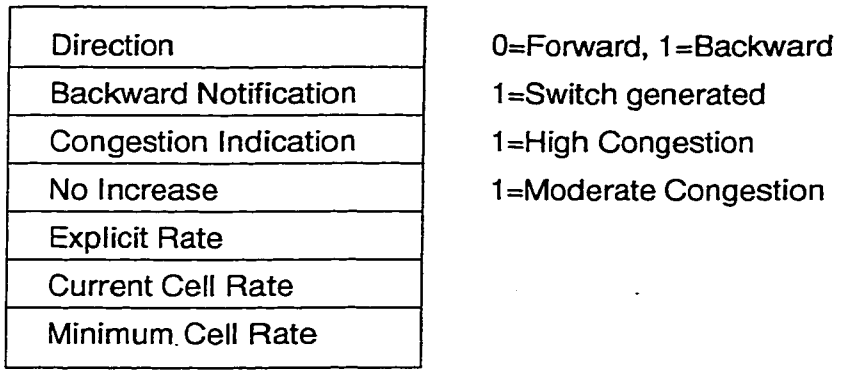

Figure 44: Resource Management (RM) Cell Fields

The Explicit Forward Congestion Indication (EFCI) bit is also found in the header of data cells. When the network is congested, it may set it to 1. When receiving date cells, the destination saves the EFCI for every data cell. When it turns around a RM cell, and if the EFCI state is set, the destination set the CI bit in the RM cell to give feedback to the source. When receiving the Rm cell, the source adjusts its ACR by considering the values of ER, CI, NI, and source parameters mentioned before [JR96].

5.2 Problem Statement

The ATM forum 4.0 provides an English description of the ABR protocol. At the same time it stops short from setting a formal specification of the behavior of the protocol entities, i.e., source, destination and switches. Over the last decade, there have been some attempts (references) to extract a formal specification out of the English description for the ABR protocol. In [LDU96], an attempt to model the ABR protocol with communicating extended finite state machines with timers and to set a formal specification for the protocol is presented. However, this presentation does not state any clear formal syntax or semantics for the discussed model.
In this thesis, we remodel the ABR Source-Destination behavior presented in [LDU96] and extend it by presenting the model of the switch using TROM formalism with parameterized events. For this, we make use of the TROMLAB environment to automate the process.

5.3 ABR Behavior

In this section, we describe the source, destination, and switch behaviors based on the rules stated in the English specification presented in the ATM Forum 4.0 and affixed in Appendix B.

5.3.1 Source End Systems Rules

Here, we explain the rules of the source behavior based on the English specification. These rules govern the behavior of the source under all the possible circumstances [JR96]. However, we do not elaborate on Source Rule 4, 11, & 13 because we do not consider out-of-rate RM cells for source behavior.

Source Rule 1: The rate at which a source can transmit at any time, namely the ACR, ranges between MCR and PCR. It can not go below MCR or higher than PCR. Mathematically, this means that
MCR <= ACR <= PCR and
Source rate <= ACR

Source Rule 2: To start a connection, the source transmits a forward RM cell at the initial rate ICR. This is done to collect the network feedback as soon as possible.
Source Rule 3: A source can transmit three types of cells: data cells, forward RM cells, and backward RM cells (corresponding to the reverse flow). The relative priority of these three kinds of cells is different at different transmission opportunities. The source issues FRM cells every Nrm-1 cells. In case of congestion, the source does not wait for the specified number of cells to be transmitted before issuing an FRM cell. This might lead to delayed feedback from the network. Therefore, the source issues an FRM cell after Trm time units have elapsed. However, this solution in itself shows the following deficiency. If the source is a low rate source, it will keep transmitting FRM cells every Trm time units. As a result, the source transmission rate of normal cells will be reduced immensely. To avoid such conflict, the transmission of consecutive FRM cells by a source is interleaved by at least Mrm other cells.
Source Rule 5: In some situations, and due to high congestion, a source might not transmit any RM cells for a duration of ADTF (ACR Decrease Time Factor: a timeout

interval parameter that can be negotiated with the network at connection setup) time units. In this case, the source cannot use its previously allocated ACR, especially if the ACR is high. Then, in order to obtain a new ACR, the source depends on the network feedback to an RM cell. In addition, the source decreases its rate to the initial cell rate (ICR). However, if a source's ACR is already below ICR, the lower rate is kept.

Source Rule 6: When the network link is broken or highly congested, the source may not receive the feedback to all the sent RM cells. As a result, the network might get overloaded with the continuous in-flow traffic. To avoid overloading the network, the sources have to reduce their transmission rates when the feedback from the network is not received after a certain time. Normally sources receive one BRM for every FRM they send. When the network is congested, BRM cells may be delayed. After Crm FRM cells are sent, if no BRM cells are received, the source considers the network congested and reduces its ACR by a factor of CDF. The two parameters Crm (missing RM cell count) and CDF (cutoff decrease factor) are set at the time of connection setup. On the other hand, BECN cells (identified by BN=1) generated by switches, during high congestion, and sent to the source are not counted as BRM cells.

Source Rule 7: When an FRM cell is transmitted, the source assigns its ACR to the CCR field of the RM cell.

Source Rule 8 & 9: The network feedback to the source includes the explicit rate (ER), the congestion indication bit (CI), and the noincrease bit (NI). The normal reaction to the network feedback is to change the ACR to the incoming ER value. In order to do so, the source has to take into account the following: The new ER is very high compared to current ACR: if the source switches to the new ER, the network will be faced with a high rate of transmission that causes sudden queues. To avoid this problem, the source is forced to increase its ACR by only a limited amount. The rate increase factor (RIF) parameter is used to indicate the maximum allowed increase the source can perform in one step. The source cannot increase its ACR by more than RIFPCR. When switches set EFCI bits in the data cell headers, they do not change the value of ER. When the destination receives these changed cells, it checks the modified EFCI bits. When sending a BRM cell, it returns the EFCI bit of the last data cell received in the CI field of the BRM cell. If the CI value is 1, it implies to the source that the network is congested and that the source should reduce

its ACR. The rate decrease factor (RDF) parameter determines the amount of ACR reduction. The No Increase (NI) bit is used to indicate moderate congestion in which case a switch could express a desired rate in the ER field. This tells the source not to increase the rate if ACR is already below the specified ER. The actions changing the value of ACR corresponding to the various combinations of values of CI and NI bits are shown in Table 13:

NI	CI	Action
0	0	ACR Min(ER, Min(ACR RIFPCR, PCR))
0	1	ACR Min(ER, ACR ACRRDF)
1	0	ACR Min(ER, ACR)
1	1	ACR Min(ER, ACR ACRRDF)

Table 13: ACR values relative to CI and NI values

Source Rule 10: the source should set the direction bit to 0 (forward). It should set the backward notification (BN) bits to 0 (source generated). It also should set the explicit rate (ER) field to the maximum rate below PCR that the source can have. It should set the current cell rate (CCR) to its current ACR. Minimum cell rate is set to the value negotiated at connection setup.
Source Rule 12: The EFCI bit must be reset whenever a data cell is transmitted.

5.3.2 Destination End System Rules

Here, we explain the rules of the destination behavior based on the English specification. These rules govern the behavior of the source under all the possible circumstances [JR96].

Destination Rule 1: The destination checks the EFCI bits on the incoming cells and store the value of the last received data cell.
Destination Rule 2: the destination turns around the forward RM cells with minimal modifications. These modifications are as follows: it changes the DIR bit to 1 as an indication that the cell is a backward RM cell. It sets the BN bit to 0 to indicate that the cell is not generated by a switch. If it has the last cell's EFCI bit set to 1, it changes the CI bit in the next BRM to 1 and it resets its stored EFCI to 0. If the

destination has internal congestion, it may act like a switch and perform one of these two options: reducing the ER or set the CI or NI bits.
Destination Rule 3 & 4: When RM cells are received by the destination, they should be returned as fast as possible. However, if the reverse ACR is not high enough to allow the fast response, BRM cells may be delayed. Under such circumstances, any out-of-date information can be discarded by the destination. This can be done in any of the following ways:

1. At the arrival of the last FRM cell, the contents of this cell are sent back directly in an out-of-rate BRM cell. In addition, the received FRM is handled and scheduled for in-rate transmission.

2. The contents of the old RM cell are sent as an out-of-rate BRM cell. Any new FRM cell is handled and scheduled for in-rate transmission.

3. The new FRM cell is handled and scheduled for in-rate transmission. Any old RM cell is simply discarded.

4. The destination makes a copy of the newly arrived FRM cell, overwrites any old cell, and schedules the two copies of the new cell for in-rate transmission.

5. The destination keeps the old cell and schedules it with the new cell for in-rate transmission.

In all of the above options, the feedback is guaranteed to flow correctly to the source. For example, the use of the out-of-rate transmission ensures that the feedback is delivered regularly under low or even zero ACR. For more details about how the destination handles old information under low ACR, we refer the readers to the Information Appendix IA.7 of the ATM forum.
Destination Rule 5: When the destination experiences high congestion, it may require that the source rate be decreased immediately without waiting for the next RM cell to load it with such feedback and turn it around. In this case, the behavior of the destination becomes similar to that of the switch when generating BECN RM cells. Also, like switch generated BECN cells, these cells may not ask a source to increase its rate by having the CI set.
Destination Rule 6: We don't elaborate on this rule because, in our model, we do not

consider out-of-rate RM cells for source behavior.

5.3.3 Switch Rules

The following is a description of the rules that govern the behavior of the switch in the network. These rules distinguish between two behaviors exhibited by the switch. The first corresponds to the in-rate cell stream while the second refers to the out-of-rate cell stream [JR96].

Switch Rule 1: In the case of congestion, the switch can notify the other entities by:

1. Setting the EFCI bit in the data cells received.
2. Setting the CI(respectively NI) bit to 1 if the congestion is high (respectively low).
3. Reducing the ER field of the FRM and/or BRM cells.

Switch Rule 2: When the switch is highly congested, it may generate its own BRM cell and transmit it to the source without waiting for an RM to arrive. In the generated BRM cell, the switch sets either CI or NI to 1 but not both simultaneously. In addition, the switch sets BN to 1 to indicate that the generated RM cell is an out-of-rate cell and the DIR bit to 1 for backward direction.
Switch Rule 3: The switch processes received cells of the same type by the order they are received. However, RM cells may be prioritized over data cells.
Switch Rule 4: For RM cells that are received by the switch, processed, and then forwarded in either direction, the switch is allowed to change the following fields: CI, NI, and ER are changed based on SWR1 while MCR is changed if it is different from the connection's MCR.
Switch Rule 5: The switch can choose to lower the ACR of an end-system to the proximity of the real transmission rate reached over the connection between the switch and the end-system. (This rule is not considered in our model for simplicity purposes).

5.4 The System Model and its Parameters in TROM

We model the ABR protocol according to the above-explained rules. We represent the model as a class diagram depicting the network structure. It includes three classes: Remote Source, Remote Destination, and Switch.
However, the behavior of both the source and destination involve sending and receiving information and feedback. This makes each end system act as a source and a destination at the same time. Therefore, the Remote Source and Remote Destination classes are composite classes that include both a source class and a destination class. Also, it is implied in the English description that a scheduler is needed to organize transmission of data, FRM, and BRM cells by the source. Therefore Source Rules 3, 5, 6, and 7 represent the rules for the scheduler which is modeled as a separate class. Hence the end system classes namely Remote Destination and Remote Source each includes three classes: source, destination, and scheduler. When the end system acts as a source, the source class is involved in initialization, and sending data cells and FRM cells via the scheduler class. When the end system acts as a destination, the destination class is involved in receiving FRM cells, turning them to BRM cells, and handing them to the source class which in turn hands them to the scheduler for transmission. The scheduler transmits the cells based on two timers: one to count the time elapsed between any two transmitted cells which is equal to 1/ACR, and one to count the time since the last transmitted FRM cell. These timers are modeled using two sets of time constraints each representing the two timers. These three entities, destination, source, and scheduler, as described in [LDU96], interface with each other via two queues: a queue of BRM cells between the source and scheduler and a queue of turned around BRM cells between the destination and source. Since in TROM, data sharing is not allowed, we model these two queues as separate TROMs that synchronize with the other classes. On the other hand, we designed the Switch class to be a composite class of three classes: Receiver, Sender, and SWQueue. The Receiver receives incoming cells and stores them in the SWQueue. The Sender retrieves the cells from SWQueue and processes them for transmission.

Figure 45 shows the block diagram to illustrate the structure of the ABR network.

Parameters: To model the ABR system, we divide the parameters into two

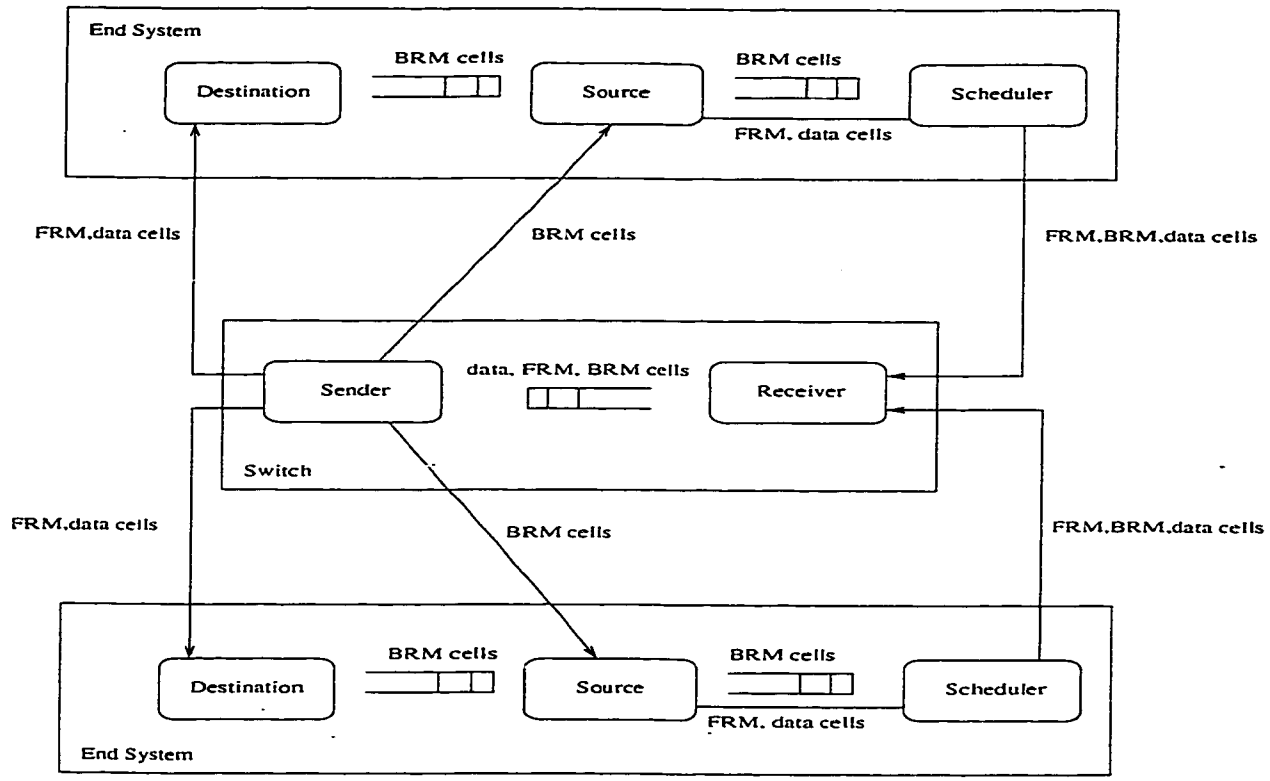

Figure 45: Block Diagram for the ABR Network

categories: dynamic and constant parameters. The dynamic parameters are the ones associated with cells and their values are changes throughout the system. These parameters are: EFCI, CI, NI, BN, DIR, and ER. The constant parameters are system parameters and are not associated with cells. They are negotiated at connection setup, and their values stay constant throughout the system. These parameters are: ADTF, ICR, MCR, Nrm, PCR, RDF, Nrm, and Trm.

Having the queues shared between the ABR classes necessitates the introduction of some new parameters to the system. The type of the cell distinguished by taking the value 1 for FRM cells, 2 for BRM cells, and 3 for data cells is represented by the parameter Type. The number of turned around BRM cells received by the destination class and queued to the source is represented by N. The number of BRM cells queued by the source to the scheduler is represented by X. Also, in the Switch, the number of cells queued by the Receiver to the Sender is represented by S. Another parameter is introduced to represent a data type sharing between the source and the scheduler classes. This parameter, Y, is the number of FRM cells sent since the last turned around BRM.

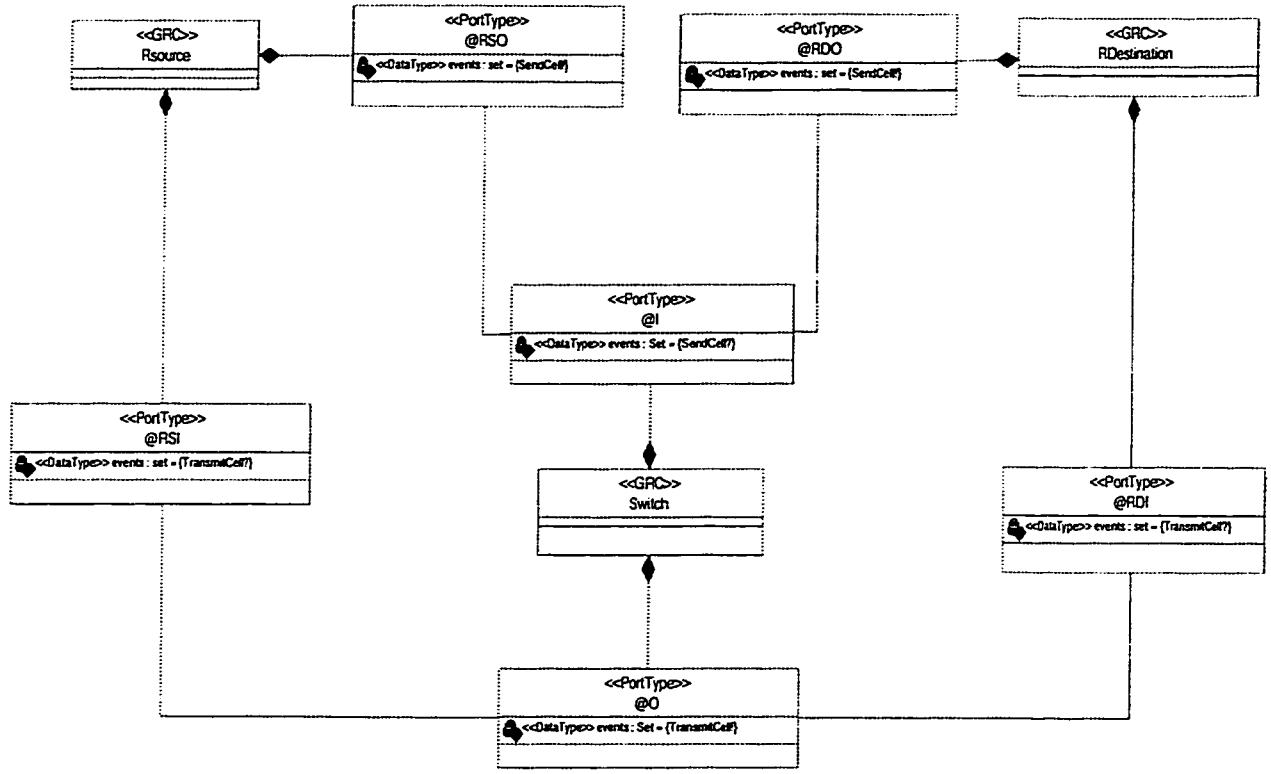

Figure 46: ABR Network Class Diagram

5.5 Rose Model of the ABR Protocol

In this section, we present the Rose model of the ABR protocol. However, this model is incomplete due to limitations of Rational Rose tool. The Rose modeler does not support the notion of composite classes as in TROM formalism. Therefore, we show the diagrams of the composing classes. Also, due to the complexity of the composing classes state diagrams, we did not model the product machine of the composite class, but we showed the behavior of each individual class.

5.5.1 Network Class Diagram

The network class diagram in Figure 46 is composed of *RSource, RDestination, and Switch* classes.

RSource GRC is an aggregate of port types *@RSI* and *@RSO*. *RDestination* GRC is an aggregate of port types *@RDI* and *@RDO*. *Switch* GRC is an aggregate of port types *@I* and *@O*. There is an association between port type *@RSO* of class *RSource* and

port type *@I* meaning that the GRC *RSource* uses the port type *@RSO* to communicate with the class *Switch* through port type *@I*. There are associations between the two port types *@RSO* and *@RSI* of class *RSource*, and the two port types *@I*and *@O* respectively meaning that the GRC *RSource* uses the port types *@RSO* and *@RSI* to communicate with the class *Switch* through port types *@I* and *@O* respectively. There are associations between the two port types *@RDO* and *@RDI* of class *RDestination*, and the two port types *@I*and *@O* respectively meaning that the GRC *RSource* uses the port types *@RDO* and *@RDI* to communicate with the class *Switch* through port types *@I* and *@O* respectively. Class *RSource* has two port types *@RSO* and *@RSI* where respectively the following events may occur: output event *SendCell* and input event *TransmitCell*. Class *RDestination* has two port types *@RDO* and *@RDI* where respectively the following events may occur: output event *SendCell* and input event *TransmitCell*.

However, these three GRCs are composite classes. In the following sections, we describe each composite class as separate class diagrams. However, Rational Rose tool does not provide a facility for modeling composite classes. Hence, it is impossible to graphically design composite classes. To introduce the concept itself in Rose and modify the tool is a substancial effort, and is not part of this thesis.

5.5.2 End Systems Class Diagrams

Figure 47 shows the class diagram that includes the composing classes of the *RSource* and *RDestination* GRCs. The diagram includes *Destination, Source, Scheduler, Queue,* and *SQueue* GRCs.

Destination GRC is an aggregate of port types em @DI and *@D*. *Source* GRC is an aggregate of port types *@SI, @S,* and *@R*. *Scheduler* GRC is an aggregate of port types em @SI and *@Sc*. *Queue* GRC is an aggregate of port types *@Q* and *@P*. *SQueue* GRC is an aggregate of port types *@M* and *@N*.

There is an association between port type *@D* of class *Destinaion* and *@P* of class *Queue* meaning that the GRC *Destination* uses the port types *@D* to communicate with the class *Queue* through port types *@P*.

There are associations between the two port types *@S* and *@R* of class *Source* and the two port types *@Q* of class *Queue* and *@M* of class *SQueue* respectively meaning

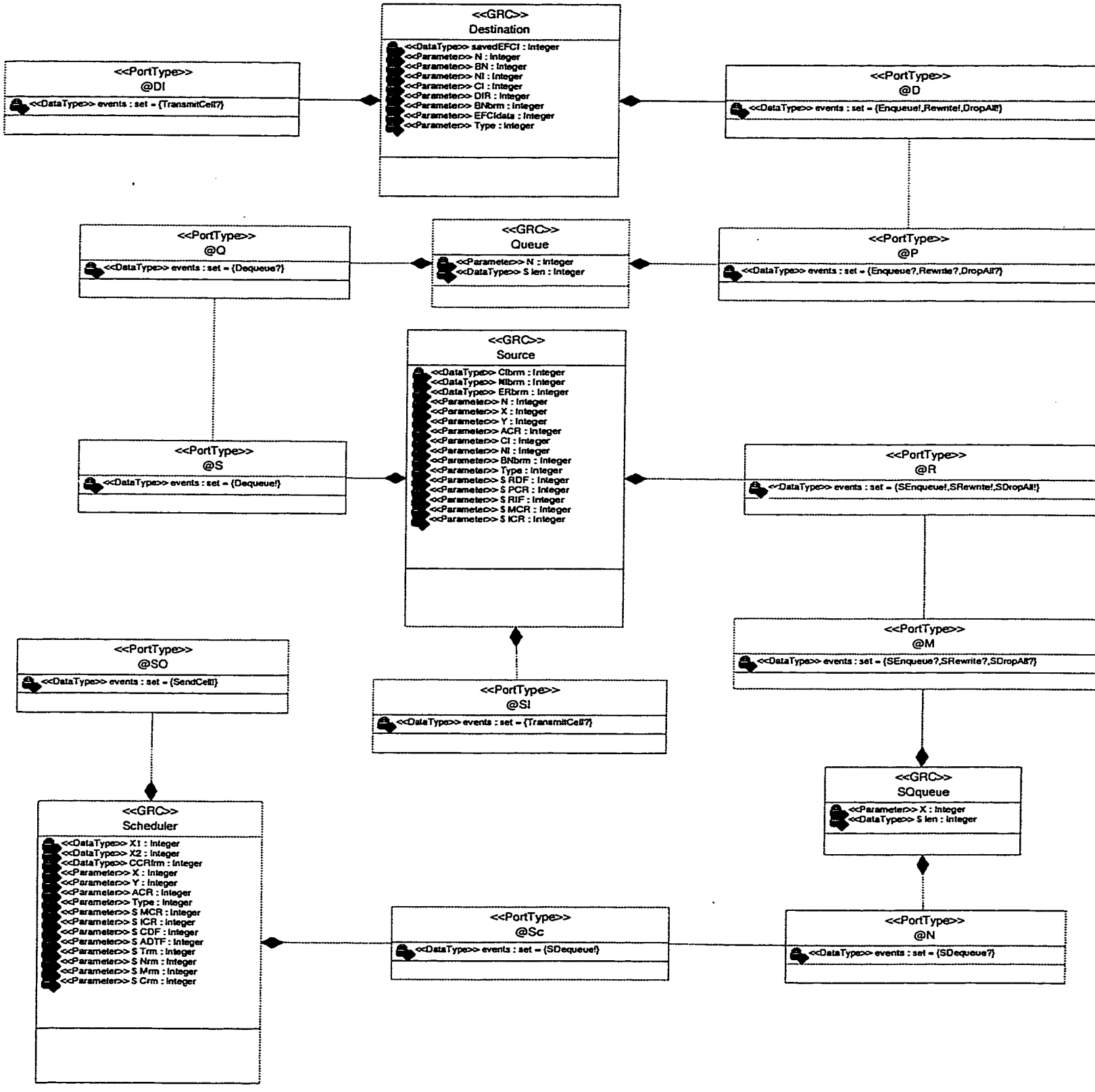

Figure 47: ABR RSource & RDestination Composite Class Diagram

that the GRC *Source* uses the port types *@S* and *@R* to communicate with the GRCs *Queue* and *SQueue* through port types *@Q* and *@M* respectively.
There is an association between port type *@Sc* of class *Scheduler* and *@N* of class *SQueue* meaning that the GRC *Scheduler* uses the port types *@D* to communicate with the GRC *SQueue* through port types *@N*.
The port types *@DI, @SI,* and *@SO* are used in the diagram to express their relations with the port types of the composing class. Through these port types the actual communication of the network occurs.
Destination GRC has two port types: *@DI* and *@D*. At port type *@DI*, the following event may occur: input event *TransmitCell*. At port type *@D*, the following events may occur: output events *Enqueue, Rewrite,* and *DropAll*.
Destination GRC has one data type attribute, *saved_EFCI* of type integer, in which the *EFCI* of the last data cell received is stored. It has the following variable parameters: *N, BN, NI, CI, DIR, BNbrm, EFCIdata,* and *Type* where *BNbrm* is the *BN* bit of the BRM to be sent and *EFCIdata* is the *EFCI* bit of the data cell received. They are all of type integer.
Source GRC has three port types: *@SI, @S,* and *@R*. At port type *@SI*, the following event may occur: input event *TransmitCell*. At port type *@S*, the following events may occur: output events *Dequeue*. At port type *@R* the following events may occur: output events em SEnqueue, SRewrite, and *SDropAll*.
Source GRC has three data type attributes, *CIbrm, NIbrm,* and*ERbrm* that correspond to *CI, NI,* and *ER* of the BRM received. They are of type integer. It has the following variable parameters: *N, X, Y, ACR, BNbrm, NI, CI,* and *Type*. They are all of type integer. It has also the following constant parameters: *RDF, PCR, RIF, MCR,* and *ICR*. They are of type integer.
Scheduler GRC has two port types: *@SO,* and *@Sc*. At port type *@SO*, the following event may occur: output event *SendCell*. At port type *@Sc*, the following event may occur: output event *SDequeue*.
Scheduler GRC has three data type attributes, *X1, X2,* and*CCRfrm* of type integer where *X1* and *X2* designate the counter of sent BRM cells and data cells respectively. *CCRfrm* corresponds to the *CCR* of the FRM to be sent. It has the following variable parameters: *X, Y, ACR,* and *Type*. They are all of type integer. It has also the following constant parameter: *MCR, ICR, CDF, ADTF, Trm, Nrm, Mrm,* and *Crm*.

They are of type integer.

Queue GRC has two port types: *@Q*, and *@P*. At port type *@Q*, the following event may occur: input event *Dequeue*. At port type *@P*, the following events may occur: input events *Enqueue*, *Rewrite*, and *DropAll*.

Queue GRC has one data type attribute, *len*, a constant of type integer designating the length of the queue. It has one variable parameter, N of type integer.

SQueue GRC has two port types: *@N*, and *@M*. At port type *@N*, the following event may occur: input event *SDequeue*. At port type *@M*, the following events may occur: input events *SEnqueue*, *SRewrite*, and *SDropAll*.

Queue GRC has one data type attribute, *len*, a constant of type integer designating the length of the queue. It has one variable parameter, X of type integer.

Destination GRC

The statechart diagram for *Destination* in shown in Figure 48.

Queue GRC

The statechart diagram for *Queue* in shown in Figure 49.

Source GRC

The statechart diagram for *Source* in shown in Figure 50.

SQueue GRC

The statechart diagram of the *SQueue* class is illustrated in Figure 51.

Scheduler GRC

The statechart diagram of the *Scheduler* class is illustrated in Figure 52.

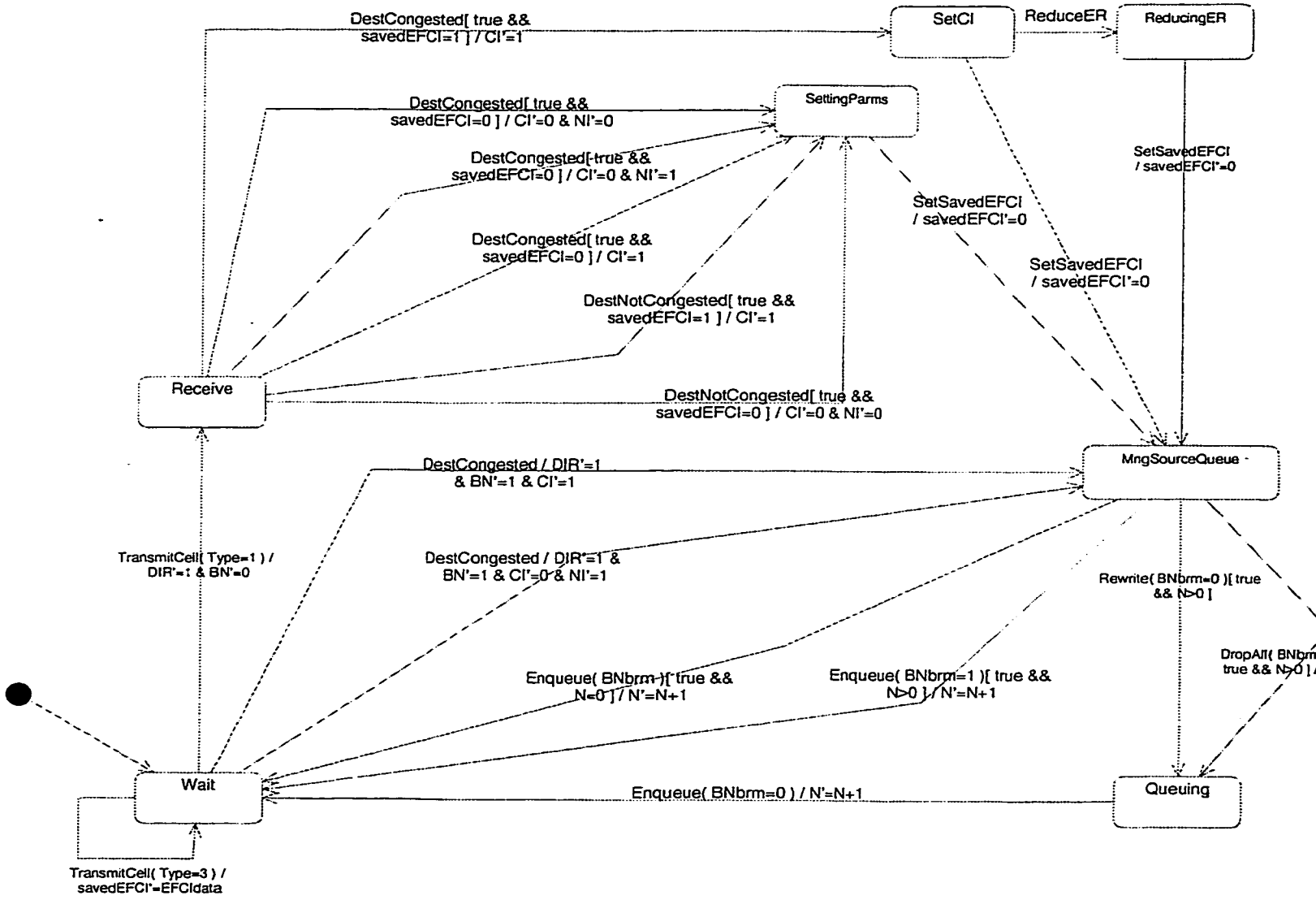

Figure 48: Destination Statechart Diagram

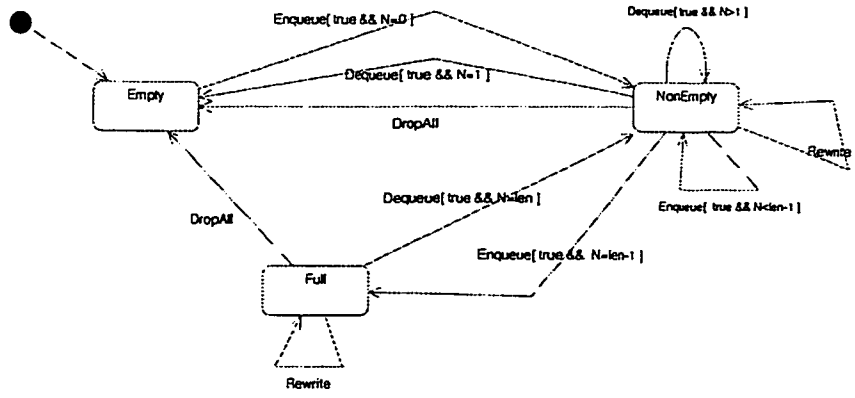

Figure 49: Queue Statechart Diagram

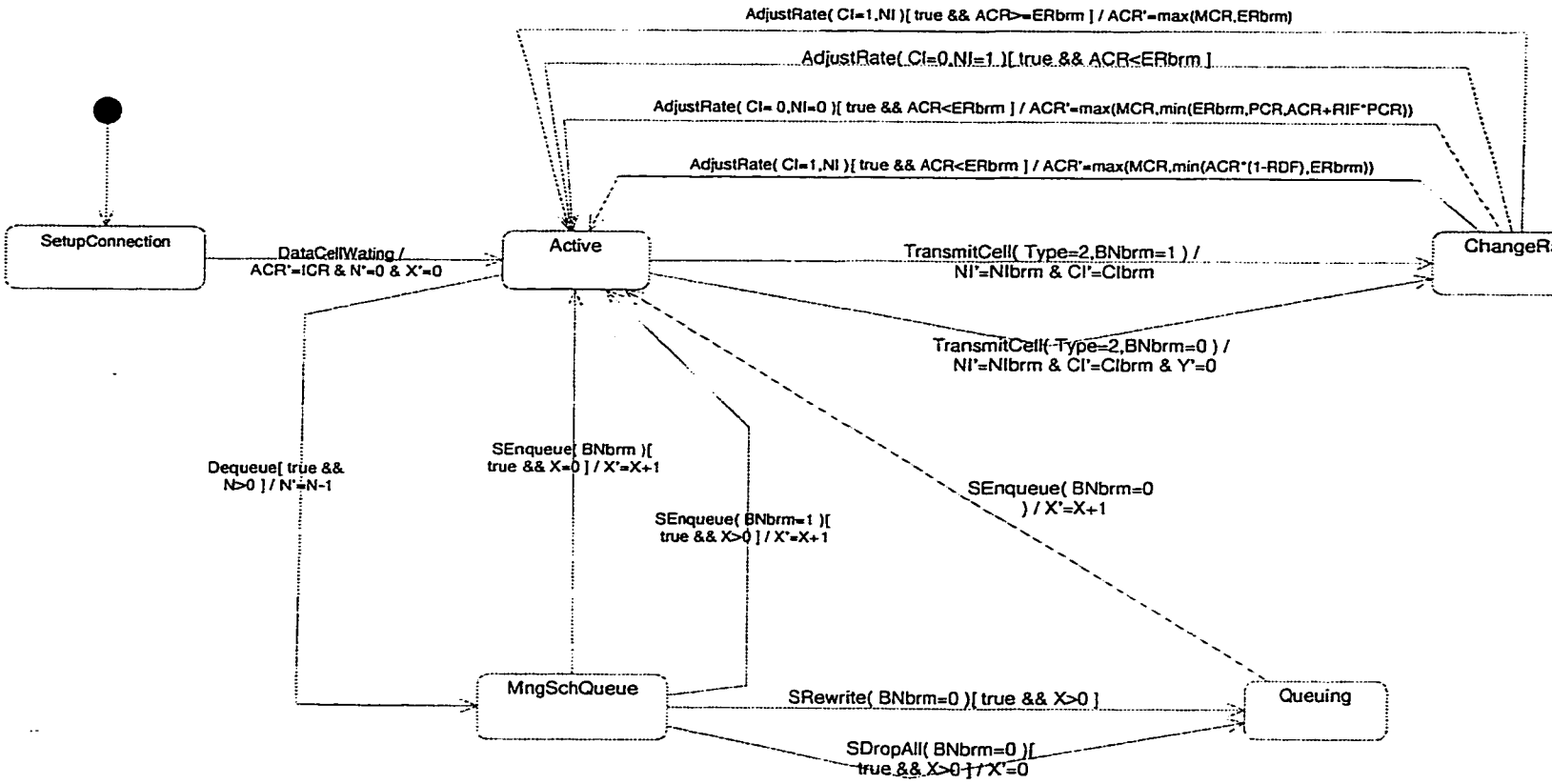

Figure 50: Source Statechart Diagram

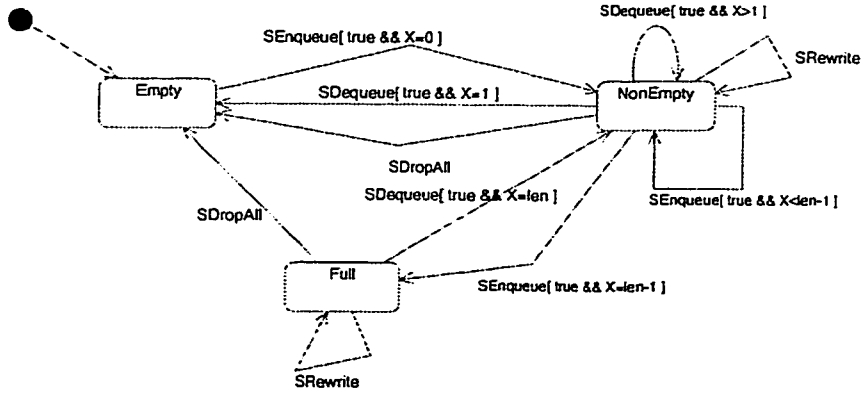

Figure 51: SQueue Statechart Diagram

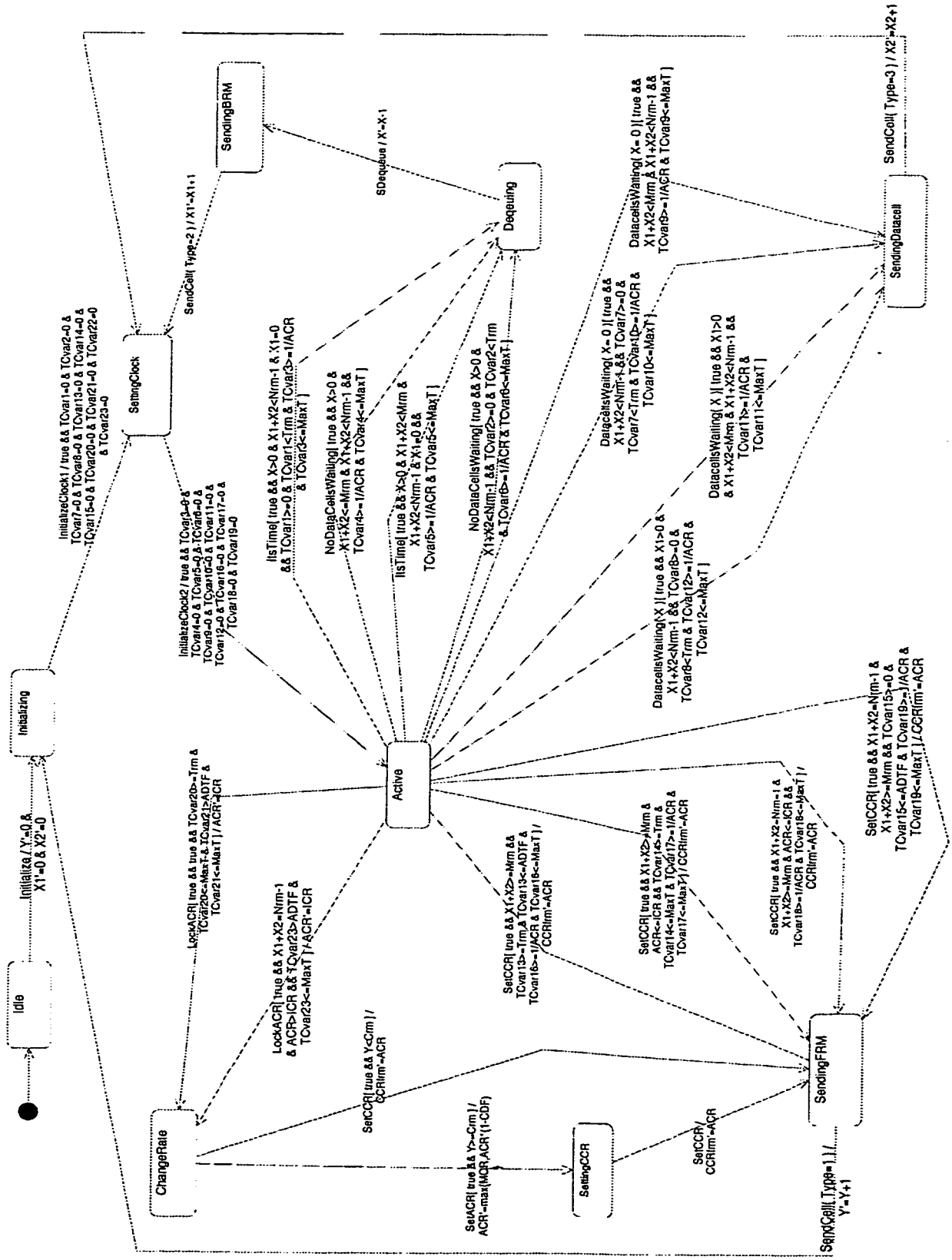

Figure 52: Scheduler Statechart Diagram

5.5.3 Switch Class Diagram

Figure 53 shows the class diagram that includes the composing classes of the *Switch* GRC. The diagram includes *Receiver, Sender,* and *SWQueue* GRCs. *Receiver* GRC is an aggregate of port types em @SWI and *@SWR*. *Sender* GRC is an aggregate of port types *@SWS* and *@SWO*. *SWQueue* GRC is an aggregate of port types *@QS* and *@QR*. There is an association between port type *@SWR* of class *Receiver* and *@QR* of class *SWQueue* meaning that the GRC *Receiver* uses the port types *@SWR* to communicate with the GRC *SWQueue* through port types *@QR*.
There is an association between port type *@SWS* of class *Sender* and *@QS* of class *SQueue* meaning that the GRC *Sender* uses the port types *@SWS* to communicate with the GRC *SWQueue* through port types *@QS*.
The port types *@SWI,* and *@SWO* are used in the diagram to express their relations with the port types of the composing class. Through these port types the actual communication of the network occurs.
Receiver GRC has two port types: *@SWI* and *@SWR*. At port type *@SWI*, the following event may occur: input event *SendCell*. At port type *@SWR*, the following event may occur: output event *Enqueue*.
Receiver GRC has two variable parameters: *S,*and *Type*. They are all of type integer.
Sender GRC has two port types: *@SWS,* and *@SWO*. At port type *@SWS*, the following event may occur: output event *Dequeue*. At port type *@SWO*, the following event may occur: output event *TransmitCell*.
Sender GRC has two variable parameters: *S,*and *Type*. They are all of type integer.
SWQueue GRC has two port types: *@QS,* and *@QR*. At port type *@QS*, the following event may occur: input event *Dequeue*. At port type *@QR*, the following event may occur: input event *Enqueue*.
Sender GRC has two variable parameters: *S*and *Type*. They are all of type integer. It has also one attribute, *Qtype*, which is a trait designating a queue data type to store the types of the cells received into the *SWQueue*.

Receiver GRC

Figure 54 illustrates the statechart diagram of the *Receiver* class.

SWQueue GRC

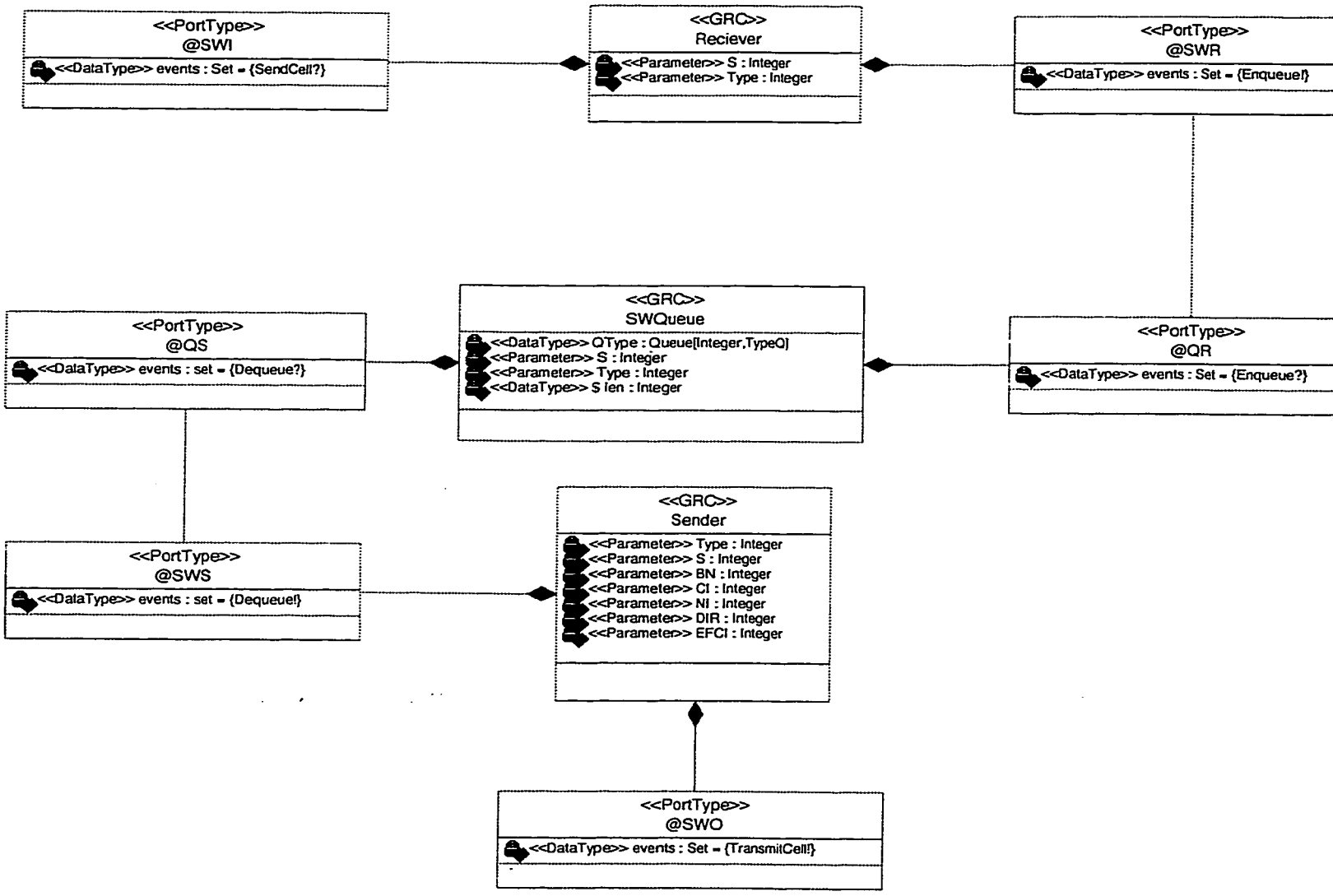

Figure 53: ABR Switch Composite Class Diagram

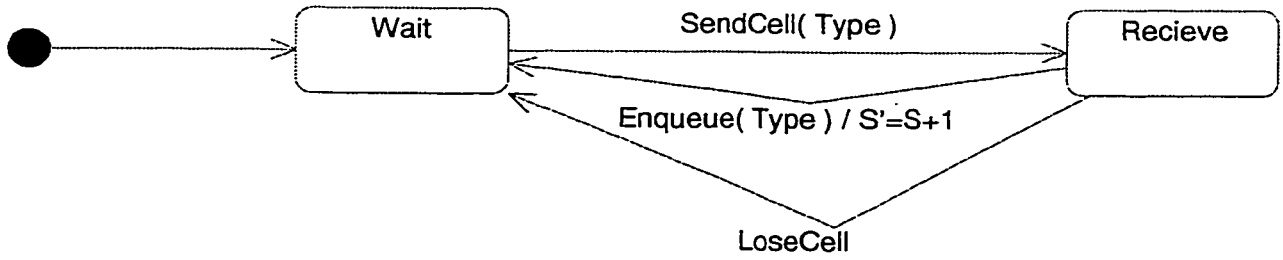

Figure 54: Receiver Statechart Diagram

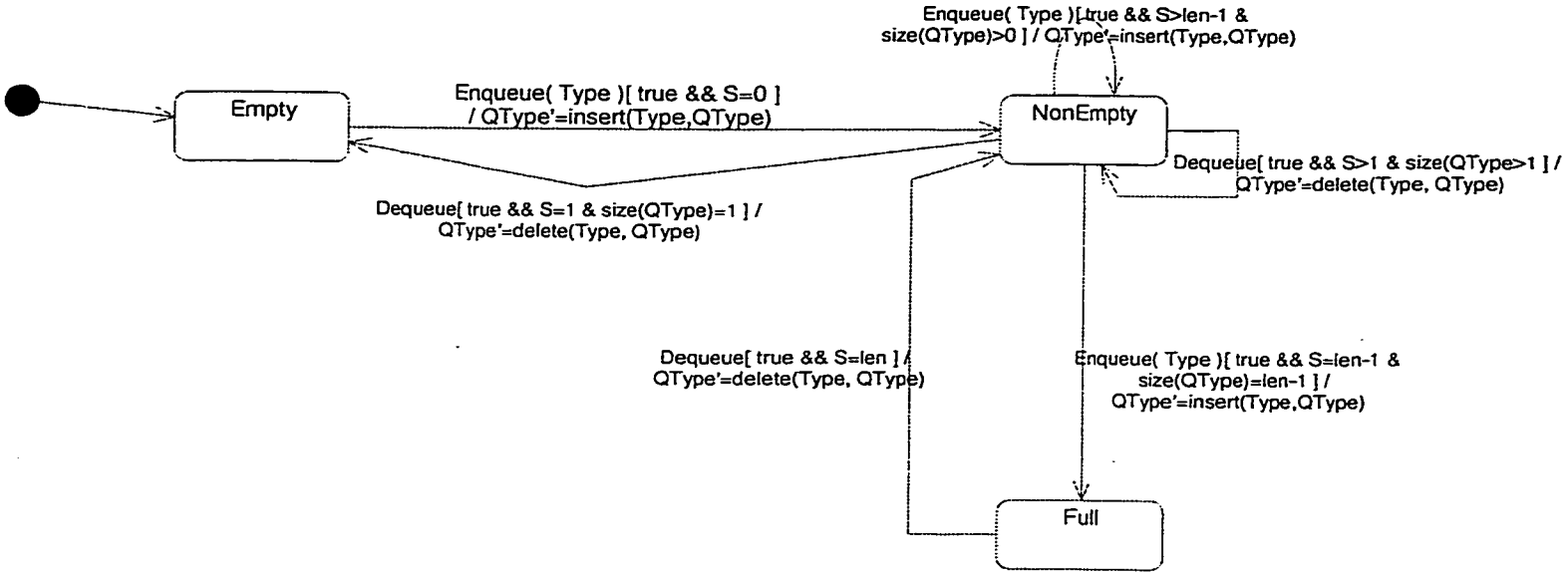

Figure 55: SWQueue Statechart Diagram

Figure 55 illustrates the statechart diagram of the *SWQueue*.

Sender GRC

Figure 56 illustrates the statechart diagram of the *Sender* class.

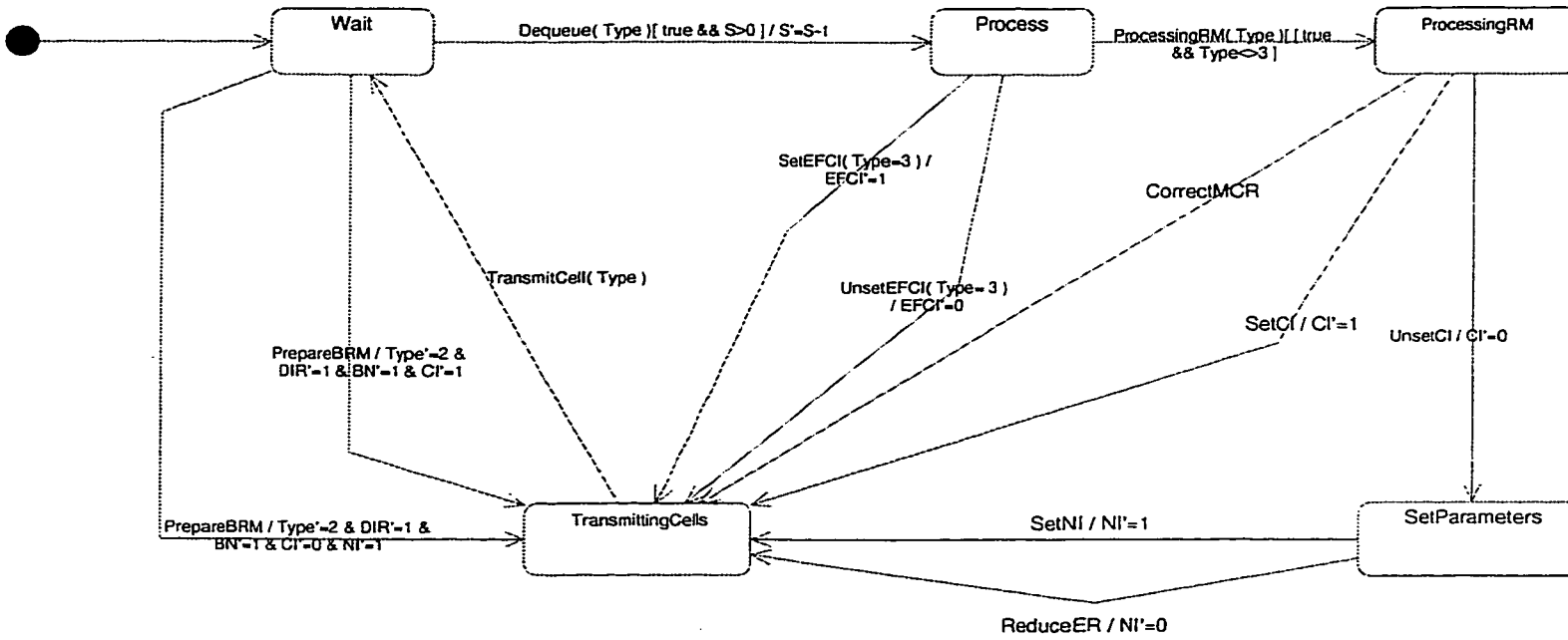

Figure 56: Sender Statechart Diagram

5.5.4 Collaboration Diagrams

In Figures 57 and 58 we show the collaboration diagrams of the End System and Switch class diagrams respectively. Since composite classes are not supported by the Rose tool, we do not show the communication between the composing classes with the composite classes.

In Figure 59 we show the collaboration diagram of the Network class diagram. However, due to the limitaton stated above we were not able to produce the subsystem configuration specification for this diagram.

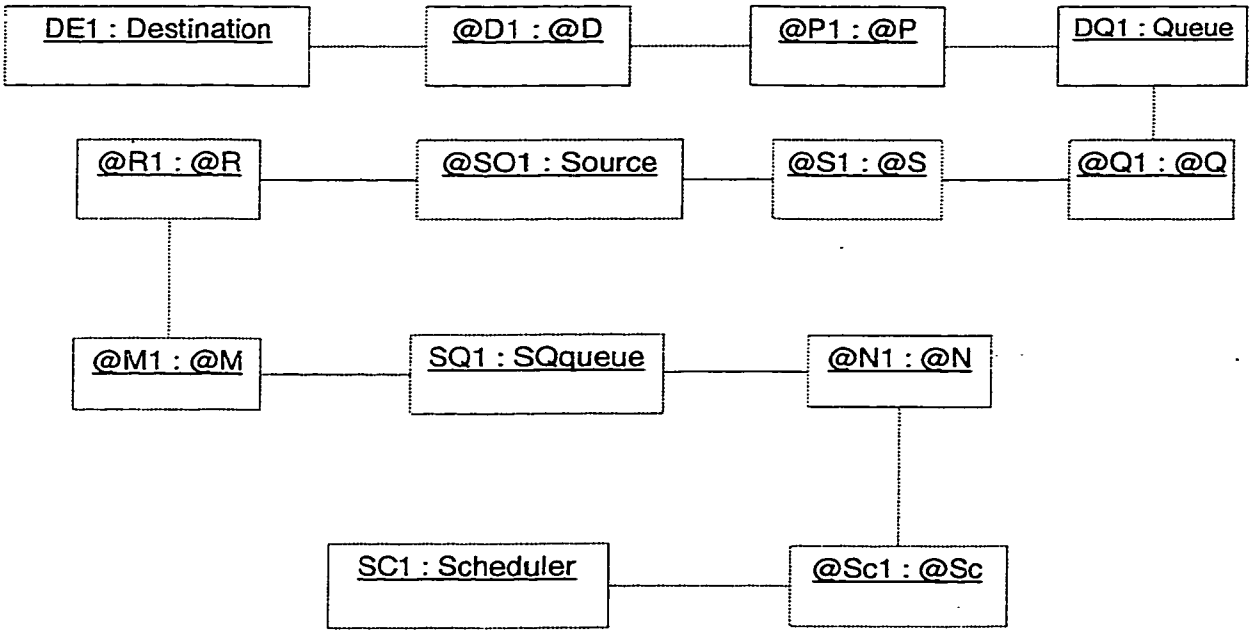

Figure 57: End System collaboration Diagram

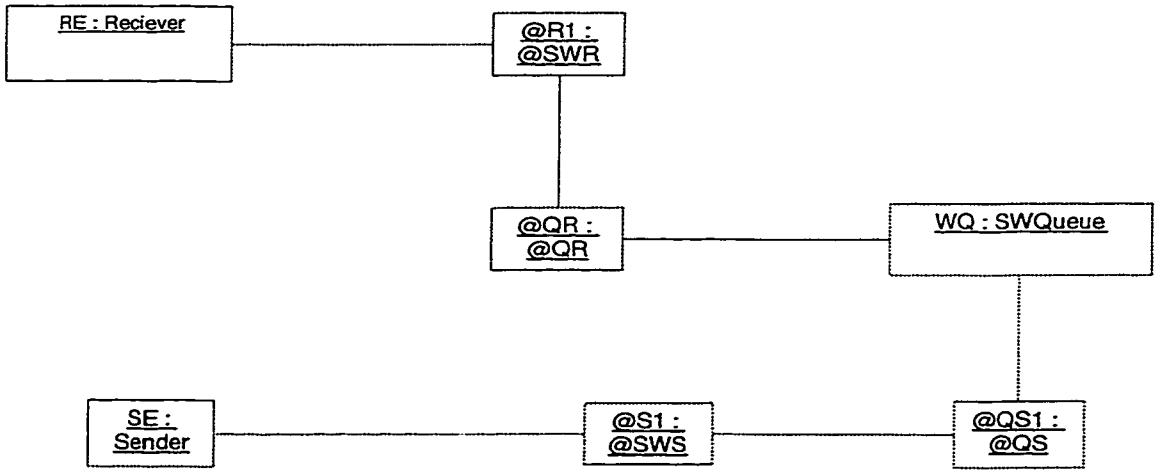

Figure 58: Switch Collaboration Diagram

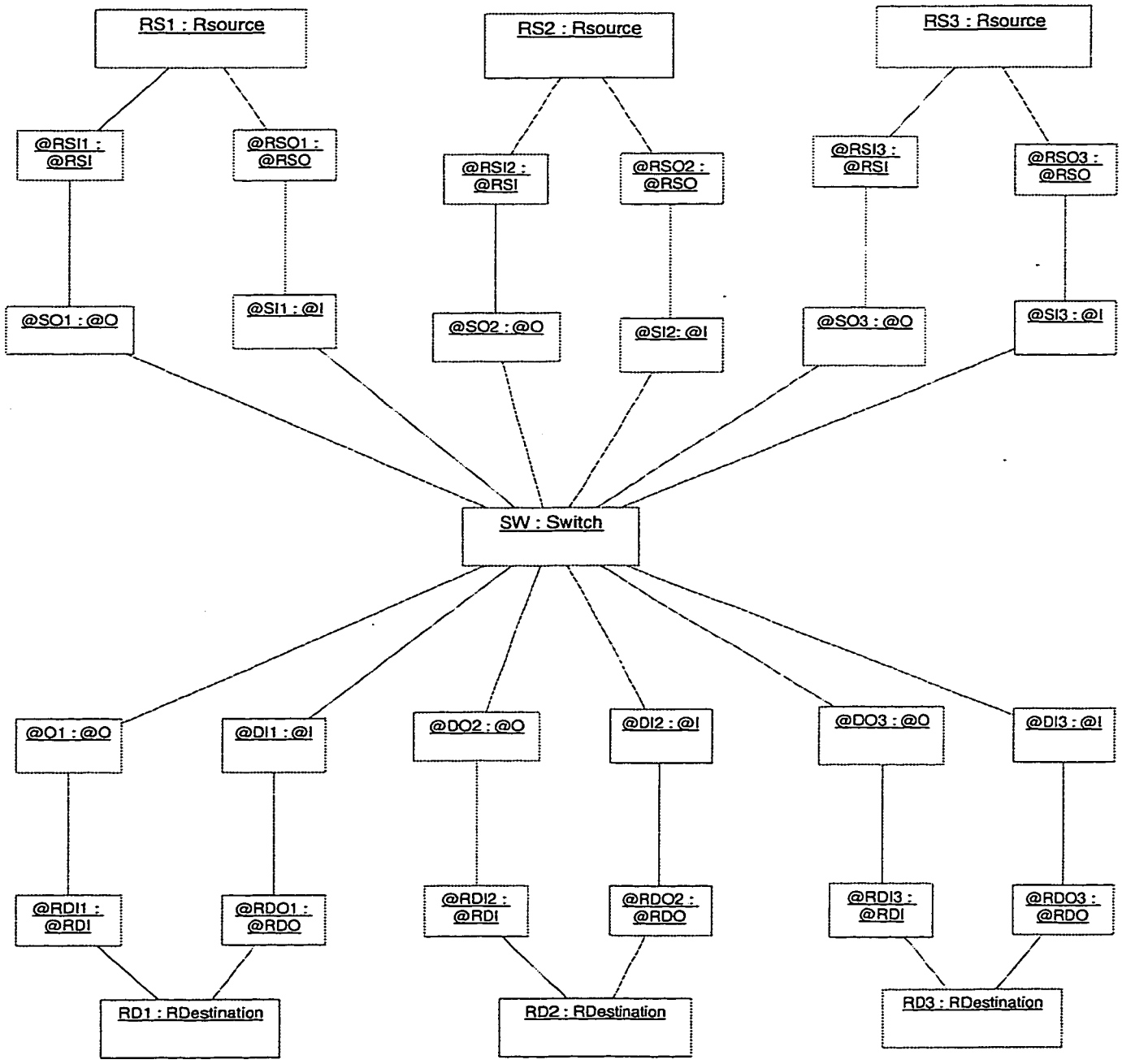

Figure 59: Network Collaboration Diagram

5.6 Formal Specifications

Figures 62,60,64,61, and 63 show the formal specifications of the classes composing the end system class.
Figures 65,67,66 show the formal specifications of the classes composing the switch class.
Figures 68, and 69 shows the subsystem configuration specification of the classes composing the End System class and the classes composing the Switch class.

Class Destination [@D, @DI]
Events: TransmitCell?@DI, DestCongested, Enqueue!@D, Rewrite!@D, DropAll!@D,
SetSavedEFCI, ReduceER, SetSavedEFCI, DestNotCongested
States: *Wait, MngSourceQueue, ReducingER, Queuing, SetCI, SettingParms, Receive
Attributes: savedEFCI:Integer;N:Integer;BN:Integer;NI:Integer;CI:Integer;
DIR:Integer;BNbrm:Integer;EFCIdata:Integer;Type:Integer Traits:
Attribute-Function: Wait → {savedEFCI, N, Type};
MngSourceQueue → {DIR, BN, CI, NI, savedEFCI, BNbrm};ReducingER → {};
Queuing → {N, BNbrm};SetCI → {CI};SettingParms → {CI, NI};Receive → {DIR, BN}
Parameter-Specifications:
DropAll: BNbrm;
Enqueue: BNbrm;
Rewrite: BNbrm;
TransmitCell: Type;
Transition-Specifications:
R1: <Wait,Wait>; TransmitCell[Type=3](true); true ⟹ savedEFCI'=EFCIdata;
R2: <Wait,Receive>; TransmitCell[Type=1](true); true
⟹ DIR/=1 & BN/=0;
R3: <Wait,MngSourceQueue>; DestCongested[ ](true); true
⟹ DIR/=1 & BN/=1 & CI/=1;
R4: <Wait,MngSourceQueue>; DestCongested[ ](true); true
⟹ DIR/=1 & BN/=1 & CI/=0 & NI/=1;
R5: <MngSourceQueue,Wait>; Enqueue[BNbrm](true); N=0 ⟹ N/=N+1;
R6: <MngSourceQueue,Wait>; Enqueue[BNbrm=1](true); N>0 ⟹ N/=N+1;
R7: <MngSourceQueue,Queuing>; Rewrite[BNbrm=0](true); N>0 ⟹true;
R8: <MngSourceQueue,Queuing>; DropAll[BNbrm=0](true); N>0 ⟹ N/=0;
R9: <ReducingER,MngSourceQueue>; SetSavedEFCI[ ](true); true
⟹ savedEFCI/=0;
R10: <Queuing,Wait>; Enqueue[BNbrm=0](true); true ⟹ N/=N+1;
R11: <SetCI,MngSourceQueue>; SetSavedEFCI[ ](true); true ⟹ savedEFCI/=0;
R12: <SetCI,ReducingER>; ReduceER[ ](true); true ⟹ true;
R13: <SettingParms,MngSourceQueue>; SetSavedEFCI[ ](true); true ⟹ savedEFCI/=0;
R14: <Receive,SetCI>; DestCongested[ ](true); savedEFCI=1 ⟹ CI/=1;
R15: <Receive,SettingParms>; DestCongested[ ](true); savedEFCI=0 ⟹ CI/=1;
R16: <Receive,SettingParms>; DestCongested[ ](true); savedEFCI=0 ⟹ CI/=0 & NI/=0;
R17: <Receive,SettingParms>; DestNotCongested[ ](true); savedEFCI=1 ⟹ CI/=1;
R18: <Receive,SettingParms>; DestNotCongested[ ](true); savedEFCI=0 ⟹ CI/=0 & NI/=0;
R19: <Receive,SettingParms>; DestCongested[ ](true); savedEFCI=0 ⟹ CI/=0 & NI/=1;
Time-Constraints:
end

Figure 60: Formal specification for GRC Destination

Class Queue [@P, @Q]
Events: Enqueue?@P, Dequeue?@Q, DropAll?@P, Rewrite?@P
States: *Empty, Full, NonEmpty
Attributes: N:Integer;len:Integer
Traits:
Attribute-Function: Empty $\rightarrow$ {};Full$\rightarrow$ {};NonEmpty$\rightarrow$ {};
Parameter-Specifications:

Transition-Specifications:

R1: <Empty,NonEmpty>; Enqueue[ ](true); N=0 $\Longrightarrow$ true;
R2: <Full,NonEmpty>; Dequeue[ ](true); N=len $\Longrightarrow$ true;
R3: <Full,Empty>; DropAll[ ](true); true $\Longrightarrow$ true;
R4: <Full,Full>; Rewrite[ ](true); true $\Longrightarrow$ true;
R5: <NonEmpty,NonEmpty>; Enqueue[ ](true); N<len-1 $\Longrightarrow$ true;
R6: <NonEmpty,Full>; Enqueue[ ](true); N=len-1 $\Longrightarrow$ true;
R7: <NonEmpty,NonEmpty>; Dequeue[ ](true); N>1 $\Longrightarrow$ true;
R8: <NonEmpty,NonEmpty>; Rewrite[ ](true); true $\Longrightarrow$ true;
R9: <NonEmpty,Empty>; Dequeue[ ](true); N=1 $\Longrightarrow$ true;
R10: <NonEmpty,Empty>; DropAll[ ](true); true $\Longrightarrow$ true;

Time-Constraints:
end

Figure 61: Formal specification for GRC Queue

Class Source [@S, @R, @SI]
Events: DataCellWating, TransmitCell?@SI, Dequeue!@S, AdjustRate, SEnqueue!@R,
SRewrite!@R, SDropAll!@R
States: *SetupConnection, Active, ChangeRate, Queuing, MngSchQueue
Attributes: CIbrm:Integer;NIbrm:Integer;ERbrm:Integer;N:Integer;X:Integer;Y:Integer;
ACR:Integer;CI:Integer;NI:Integer;BNbrm:Integer;Type:Integer;RDF:Integer;PCR:Integer
RIF:Integer;MCR:Integer;ICR:Integer
Traits:
Attribute-Function: SetupConnection → {};Active → {ACR, N, X, Type, BNbrm};
ChangeRate → {NI, CI, Y}:Queuing → {X, BNbrm};MngSchQueue → {N, BNbrm};
Parameter-Specifications:
AdjustRate: CI, NI;
SDropAll: BNbrm;
SEnqueue: BNbrm;
SRewrite: BNbrm;
TransmitCell: Type, BNbrm;
Transition-Specifications:
R1: <SetupConnection,Active>; DataCellWating[ ](true); true
⟹ ACR/=ICR & N/=0 & X/=0;
R2: <Active,ChangeRate>; TransmitCell[Type=2,BNbrm=1](true); true
⟹ NI/=NIbrm & CI/=CIbrm;
R3: <Active,MngSchQueue>; Dequeue[ ](true); N>0 ⟹ N/=N-1;
R4: <Active,ChangeRate>; TransmitCell[Type=2,BNbrm=0](true); true
⟹ NI/=NIbrm & CI/=CIbrm& Y/=0;
R5: <ChangeRate,Active>; AdjustRate[CI=1,NI](true); ACR<ERbrm
⟹ ACR=/max(MCR,min(ACR(1-RDF),ERbrm));
R6: <ChangeRate,Active>; AdjustRate[CI=0,NI=0](true); ACR<ERbrm
⟹ ACR/=max(MCR,min(ERbrm,PCR,ACR+RIF*PCR));
R7: <ChangeRate,Active>; AdjustRate[CI=0,NI=1](true); ACR<ERbrm ⟹ true;
R8: <ChangeRate,Active>; AdjustRate[CI=1,NI](true); ACR>=ERbrm
⟹ ACR/=max(MCR,ERbrm);
R9: <Queuing,Active>; SEnqueue[BNbrm=0](true); true ⟹ X=/X+1;
R10: <MngSchQueue,Active>; SEnqueue[BNbrm](true); X=0 ⟹ X/=X+1;
R11: <MngSchQueue,Active>; SEnqueue[BNbrm=1](true); X>0 ⟹ X/=X+1;
R12: <MngSchQueue,Queuing>; SRewrite[BNbrm=0](true); X>0 ⟹ true;
R13: <MngSchQueue,Queuing>; SDropAll[BNbrm=0](true); X>0 ⟹ X/=0;
Time-Constraints:
end

Figure 62: Formal specification for GRC Source

Class SQqueue [@P, @Q]
Events: Enqueue?@P, Dequeue?@Q, DropAll?@P, Rewrite?@P
States: *Empty, Full, NonEmpty
Attributes: X:Integer;len:Integer
Traits:
Attribute-Function: Empty $\rightarrow$ {};Full$\rightarrow$ {};NonEmpty$\rightarrow$ {};
Parameter-Specifications:

Transition-Specifications:
R1: <Empty,NonEmpty>; SEnqueue[](true); X=0 $\Longrightarrow$ true;
R2: <Full,NonEmpty>; SDequeue[](true); X=len $\Longrightarrow$ true;
R3: <Full,Empty>; SDropAll[](true); true $\Longrightarrow$ true;
R4: <Full,Full>; SRewrite[](true); true $\Longrightarrow$ true;
R5: <NonEmpty,NonEmpty>; SEnqueue[](true); X<len-1 $\Longrightarrow$ true;
R6: <NonEmpty,Full>; SEnqueue[ ](true); X=len-1 $\Longrightarrow$ true;
R7: <NonEmpty,NonEmpty>; SDequeue[ ](true); X>1 $\Longrightarrow$ true;
R8: <NonEmpty,NonEmpty>; SRewrite[ ](true); true $\Longrightarrow$ true;
R9: <NonEmpty,Empty>; SDequeue[ ](true); N=1 $\Longrightarrow$ true;
R10: <NonEmpty,Empty>; SDropAll[ ](true); true $\Longrightarrow$ true;
Time-Constraints:
end

Figure 63: Formal specification for GRC SQueue

Class Scheduler [@Sc, @SO, @DO]
Events: Initialize, LockACR, NoDataCellsWaiting, ItsTime, DatacellsWaiting, SetCCR, SetACR, SendCell!@DO, SDequeue!@Sc, InitializeClock1, InitializeClock2
States: *Idle, Active, ChangeRate, SendingDatacell, Deqeuing, SendingBRM, SettingACR, Initializing, SettingClock, SettingCCR, SendingFRM
Attributes: X1:Integer;X2:Integer;CCRfrm:Integer;X:Integer;Y:Integer;ACR:Integer; Type:Integer;MCR:Integer;ICR:Integer;CDF:Integer;ADTF:Integer;Trm:Integer; Nrm:Integer;Mrm:Integer;Crm:Integer
Traits:
Attribute-Function: Idle → {};Active → {X};ChangeRate → {ACR};SendingDatacell → {Type}; Deqeuing → {};SendingBRM → {X, Type};SettingACR → {};Initializing → {Y, X1, X2}; SettingClock → {X2, X1};SettingCCR → {ACR};SendingFRM → {CCRfrm, Type};
Parameter-Specifications:
DatacellsWaiting: X;
SendCell: Type;
Transition-Specifications:
R1: <Idle,Initializing>; Initialize [ ](true); true
⟹ Y'=0 & X1'=0 & X2'=0;
R2: <Active,ChangeRate>; LockACR[](true); X1+X2=Nrm-1 & ACR>ICR
⟹ ACR'=ICR;
R3: <Active,Deqeuing>; NoData_CellsWaiting[ ](true); X>0 & X1+X2<Nrm-1
⟹ true;
R4: <Active,Deqeuing>; NoData_CellsWaiting[ ](true); X>0 & X1+X2<=Mrm & X1+X2<Nrm-1 ⟹ true;
R5: <Active,Deqeuing>; ItsTime[ ](true); X>0 & X1+X2<Nrm-1 & X1=0
⟹ true;
R6: <Active,Deqeuing>; ItsTime[ ](true); X>0 & X1+X2<Mrm & X1+X2<Nrm-1 & X1=0 ⟹ true;
R7: <Active,SendingDatacell>; DatacellsWaiting[X](true); X1>0 & X1+X2<Nrm-1
⟹ true;
R8: <Active,SendingDatacell>; DatacellsWaiting[X](true); X1>0 & X1+X2<Mrm & X1+X2<Nrm-1 ⟹ true;
R9: <Active,SendingDatacell>; DatacellsWaiting[X=0](true); X1+X2<Nrm-1
⟹ true;
R10: <Active,SendingDatacell>; DatacellsWaiting[X=0](true); X1+X2<Mrm & X1+X2<Nrm-1 ⟹ true;
R11: <Active,ChangeRate>; LockACR[ ](true); true ⟹ ACR'=ICR;
R12: <Active,SendingFRM>; SetCCR[ ](true); X1+X2>=Mrm
⟹ CCRfrm'=ACR;
R13: <Active,SendingFRM>; SetCCR[ ](true); X1+X2>=Mrm & ACR<=ICR
⟹ CCRfrm'=ACR;
R14: <Active,SendingFRM>; SetCCR[ ](true); X1+X2-Nrm=1 & X1+X2>=Mrm & ACR<=ICR ⟹ CCRfrm'=ACR;

R15: <Active,SendingFRM>; SetCCR[](true); X1+X2=Nrm-1 & X1+X2>=Mrm
$\Longrightarrow$ CCRfrm/=ACR;
R16: <ChangeRate,SettingCCR>; SetACR[ ](true); Y>=Crm
$\Longrightarrow$ ACR/=max(MCR,ACR(1-CDF));
R17: <ChangeRate,SendingFRM>; SetCCR[ ](true); Y<Crm $\Longrightarrow$ CCRfrm/=ACR;
R18: <SendingDatacell,SettingClock>; SendCell[Type=3](true); true $\Longrightarrow$ X2/=X2+1;
R19: <Deqeuing,SendingBRM>; SDequeue[ ](true); true $\Longrightarrow$ X/=X-1;
R20: <SendingBRM,SettingClock>; SendCell[Type=2](true); true $\Longrightarrow$ X1/=X1+1;
R21: <Initializing,SettingClock>; InitializeClock1[ ](true); true $\Longrightarrow$ true;
R22: <SettingClock,Active>; InitializeClock2[ ](true); true $\Longrightarrow$ true;
R23: <SettingCCR,SendingFRM>; SetCCR[ ](true); true $\Longrightarrow$ CCRfrm/=ACR;
R24: <SendingFRM,Initializing>; SendCell[Type=1](true); true $\Longrightarrow$ Y/=Y+1;

Time-Constraints:

TCvar23: R21, LockACR, (ADTF, MaxT], {};
TCvar2: R21, NoDataCellsWaiting, [0, Trm), {};
TCvar6: R22, NoDataCellsWaiting, [1/ACR, MaxT], {};
TCvar4: R22, NoDataCellsWaiting, [1/ACR, MaxT], {};
TCvar1: R21, ItsTime, [0, Trm), {};
TCvar3: R22, ItsTime, [1/ACR, MaxT], {};
TCvar5: R22, ItsTime, [1/ACR, MaxT], {};
TCvar8: R21, DatacellsWaiting, [0, Trm), {};
TCvar12: R22, DatacellsWaiting, [1/ACR, MaxT], {};
TCvar11: R22, DatacellsWaiting, [1/ACR, MaxT], {};
TCvar7: R21, DatacellsWaiting, [0, Trm), {};
TCvar10: R22, DatacellsWaiting, [1/ACR, MaxT], {};
TCvar9: R22, DatacellsWaiting, [1/ACR, MaxT], {};
TCvar20: R21, LockACR, [Trm, MaxT], {};
TCvar21: R21, LockACR, (ADTF, MaxT], {};
TCvar13: R21, SetCCR, [Trm, ADTF], {};
TCvar16: R22, SetCCR, [1/ACR, MaxT], {};
TCvar14: R21, SetCCR, [Trm, MaxT], {};
TCvar17: R22, SetCCR, [1/ACR, MaxT], {};
TCvar18: R22, SetCCR, [1/ACR, MaxT], {};
TCvar15: R21, SetCCR, [0, ADTF], {};
TCvar19: R22, SetCCR, [1/ACR, MaxT], {};

end

Figure 64: Formal specification for GRC Scheduler

Class Reciever [@SWR, @SWI]
Events: SendCell?@SWI, Enqueue!@SWR, LoseCell
States: *Wait, Recieve
Attributes: S:Integer;Type:Integer
Traits:
Attribute-Function: Wait → {S, Type};Recieve → {Type};
Parameter-Specifications:
    Enqueue: Type;
    SendCell: Type;
Transition-Specifications:
    R1: <Wait,Recieve>; SendCell[Type](true); true ⟹ true;
    R2: <Recieve,Wait>; Enqueue[Type](true); true ⟹ S'=S+1;
    R3: <Recieve,Wait>; LoseCell[ ](true); true ⟹ true;
Time-Constraints:
end

Figure 65: Formal specification for GRC Receiver

Class SWQueue [@QR, @QS]
Events: Enqueue?@QR, Dequeue?@QS
States: *Empty, NonEmpty, Full
Attributes: QType:TypeQ;S:Integer;Type:Integer;len:Integer
Traits: Queue[Integer,TypeQ]
Attribute-Function: Empty -¿ QType, Type;NonEmpty → {QType, Type};Full → {QType};
Parameter-Specifications:
    Enqueue: Type;
Transition-Specifications:
    R1: <Empty,NonEmpty>; Enqueue[Type](true); S=0 ⟹ Qtype/=insert(Type,QType);
    R2: <NonEmpty,Empty>; Dequeue[ ](true); S=1 & size(QType)=1
        ⟹QType/=delete(Type,QType);
    R3: <NonEmpty,NonEmpty>; Enqueue[Type](true); S>len-1 & size(QType)>0
        ⟹ QType/=insert(Type,QType);
    R4: <NonEmpty,Full>; Enqueue[Type](true); S=len-1 & size(QType)=len-1
        ⟹ QType/=insert(Type,QType);
    R5: <NonEmpty,NonEmpty>; Dequeue[ ](true); S>1 & size(QType>1
        ⟹ QType/=delete(Type,QType);
    R6: <Full,NonEmpty>; Dequeue[ ](true); S=len ⟹ QType/=delete(Type,QType);
Time-Constraints:
end

Figure 66: Formal specification for GRC SWQueue

Class Sender [@SWS, @SWO]
Events: Dequeue!@SWS, PrepareBRM, UnsetEFCI, SetEFCI, ProcessingRM, SetNI,
ReduceER, TransmitCell!@SWO, UnsetCI, SetCI, CorrectMCR
States: *Wait, Process, SetParameters, TransmittingCells, ProcessingRM
Attributes: Type:Integer;S:Integer;BN:Integer;CI:Integer;NI:Integer;DIR:Integer;EFCI:Integer
Traits:
Attribute-Function: Wait → {Type};Process → {S, Type};SetParameters → {CI};
TransmittingCells → {Type, DIR, BN, CI, NI, EFCI};ProcessingRM → {};
Parameter-Specifications:
Dequeue: Type;
ProcessingRM: Type;
SetEFCI: Type;
TransmitCell: Type;
UnsetEFCI: Type;
Transition-Specifications:
R1: <Wait,Process>; Dequeue[Type](true); S>0 ⟹ S/=S-1;
R2: <Wait,TransmittingCells>; PrepareBRM[ ](true); true
⟹ Type/=2 & DIR/=1 & BN/=1 & CI/=1;
R3: <Wait,TransmittingCells>; PrepareBRM[ ](true); true
⟹ Type/=2 & DIR/=1 & BN/=1 & CI/=0 & NI/=1;
R4: <Process,TransmittingCells>; UnsetEFCI[Type=3](true); true ⟹ EFCI/=0;
R5: <Process,TransmittingCells>; SetEFCI[Type=3](true); true ⟹ EFCI/=1;
R6: <Process,ProcessingRM>; ProcessingRM[Type](true); Type<>3 ⟹ true;
R7: <SetParameters,TransmittingCells>; SetNI[ ](true); true ⟹ NI/=1;
R8: <SetParameters,TransmittingCells>; ReduceER[ ](true); true ⟹ NI/=0;
R9: <TransmittingCells,Wait>; TransmitCell[Type](true); true ⟹ true;
R10: <ProcessingRM,SetParameters>; UnsetCI[ ](true); true ⟹ CI/=0;
R11: <ProcessingRM,TransmittingCells>; SetCI[ ](true); true ⟹ CI/=1;
R12: <ProcessingRM,TransmittingCells>; CorrectMCR[ ](true); true ⟹ true;
Time-Constraints:
end;

Figure 67: Formal specification for GRC Sender

```
SCS EndSystem
        Includes:
        Instantiate:
                DE1::Destination[@D:1];
                SC1::Scheduler[@Sc:1];
                DQ1::Queue[@P:1, @Q:1];
                SQ1::SQqueue[@M:1, @N:1];
                @SO1::Source[@S:1, @R:1];
        Configure:
                DQ1.@P1:@P ↔ DE1.@D1:@D;
                @SO1.@S1:@S ↔ DQ1.@Q1:@Q;
                SC1.@Sc1:@Sc ↔ SQ1.@N1:@N;
                SQ1.@M1:@M ↔ @SO1.@R1:@R;
end
```

Figure 68: SCS for End System class

```
SCS Switch
        Includes:
        Instantiate:
                RE::Reciever[@SWR:1];
                SE::Sender[@SWS:1];
                WQ::SWQueue[@QR:1, @QS:1];
        Configure:
                WQ.@QR:@QR ↔ RE.@R1:@SWR;
                WQ.@QS1:@QS ↔ SE.@S1:@SWS;
end
```

Figure 69: SCS for Switch class

Chapter 6

Verification and Validation of the ABR Protocol

In this Chapter, we model the ABR protocol using the Specification and Description Language (SDL) supported by ObjectGEODE tool where we verify and validate the ABR system. We first present an overview on SDL and its notation. Then, we show the mapping between TROM notation and SDL notation. Afterwards, we present the mapped ABR protocol model from TROM into SDL. Finally, we analyze the model by checking it against some correctness properties.

6.1 SDL - An Overview

SDL is a standard language to specify and describe systems. It is both a design and implementation language. An SDL specification consists of the following components [TLL99], [MD00]:

- **Structure:** it consists of four main hierarchies - a *system, block, process,* and *procedure* hierarchy. The system/block hierarchy is a static description of the system. An SDL system specification is modeled as a structure of blocks that can be decomposed into processes. Each process is a hierarchical state machine composed of simple and/or compound states. Each substate of a compound state is presented as a procedure.
- **Communication:** it is achieved via *signals* with optional signal parameters and *channels*. A basic communication mechanism that SDL adopts is the use

of asynchronous signals (with optional parameters). It uses the parameters to interchange the information between the processes of a system and between the system itself and its environment. The interface between blocks and processes is achieved by means of channels. Channels can be uni-directional or bi-directional. Input signals to a process are stored in a queue before reading them. This queue, a FIFO, respects the order in which the signals are received.

- **Behavior:** the behavior of the system is described in terms of processes. A process is expressed as an extended finite state machine. A process is composed of a set of states and transitions connecting them. A transition is a set of actions: local actions, procedure calls, timer set and reset, signal output, etc. When the process consumes a signal from its input queue, if the signal is not expected in the state (does not trigger any transition), it is implicitly consumed by the process and is simply discarded and lost, and the process remains in the same state. To prevent the resulting signal loss, SDL provides a *save* construct. This construct is used to keep a signal that is not removed nor consumed by the state in the queue. These saved signals are kept for future consumption.
  In SDL, time is specified using two notions: *now* and *timers*. *now*, the current time value, represents the global timer of the system. A *timer*, uniquely defined, can be set to expire within a certain amount of SDL time units. When the timer expires, the process receives a signal with the timer name. It is processed the same way as other received signals. A timer-based transition has to reach the destination state before the expiration of the timer. Otherwise, the expiration signal will be lost unless it was saved in the last state. Tow constructs are used to express the time in SDL: *now*, and *set(reset)* of *timers*. A process is uniquely identified using a process id (PID). Whenever different processes are communicating or one signal traverses through different channels, PID is used. Four predefined process identifiers are available in SDL: *self, sender, offspring,* and *parent*.

- Data: In SDL, data is described using abstract data types (ADT). An ADT is defined as a *sort* with no specified data structure. It is represented as a set of values, set of allowed operations, and a set of equations defining the operations. The following are predefined sorts in SDL: integer, real, natural, boolean, character, duration, time, charstring, and PID.

In Figure 70 and Figure 71 we show an example specification for a system *S* [MD00]. We show the static and dynamic behavior of the system. The static behavior is a structure of two blocks *B1* and *B2*. They communicate via channel *C* with signals *a*, *b*, and *c*. *B1* communicates with the environment via *C1* with signals *a* and *d*. Blocks *B1* and *B2* are decomposed into processes *P1* and processes *P2* and *P3* respectively. Processes communicate via channels inside the same block (*P2* and *P3* communicate via channel *S1*). Their communication with the other blocks is conveyed by connecting the inside channels with the channels connecting the blocks themselves.

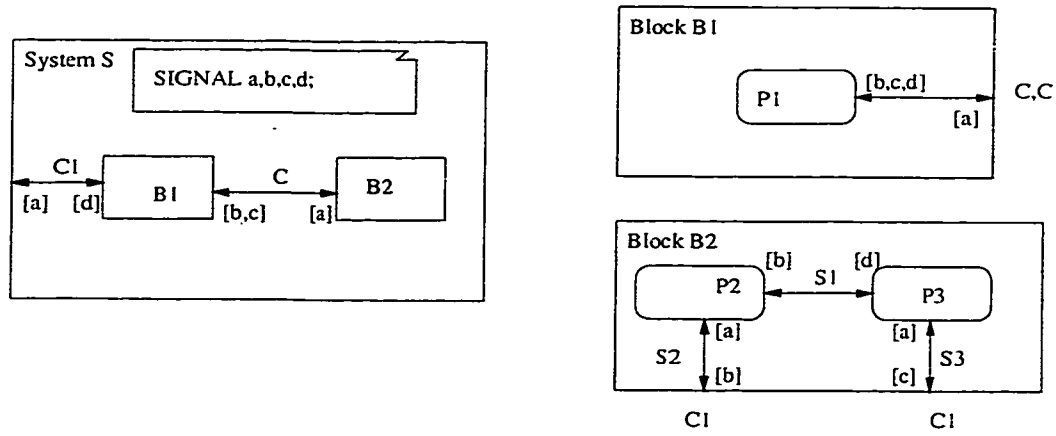

Figure 70: Example of SDL Architecture

The dynamic behavior of the system *S* is represented the processes *P1*, *P2*, and *P3*, as shown in Figure 71. The behavior of the system is defined as the parallel composition of the three processes. For instance, process *P2* moves from the *start* state into *idle* state where it expects signal *c*. If it expects signal *b* while in state *idle*, it will save it for future usage. *P2* moves from state *idle* to state *S1* by consuming the signal *c* from its input queue, performing the action (I3:=SENDER). In *S1*, process *P2* consumes signal *b*, performs the task (I1:=Sender), and sends output signal *d* to processes whose *ids* are *I1* and *I3*.

6.2 Mapping TROM Specification into SDL Implementation

A mapping from TROM specification to SDL language was presented in [MD00]. Different notations are adopted in TROM and SDL languages. To translate TROM

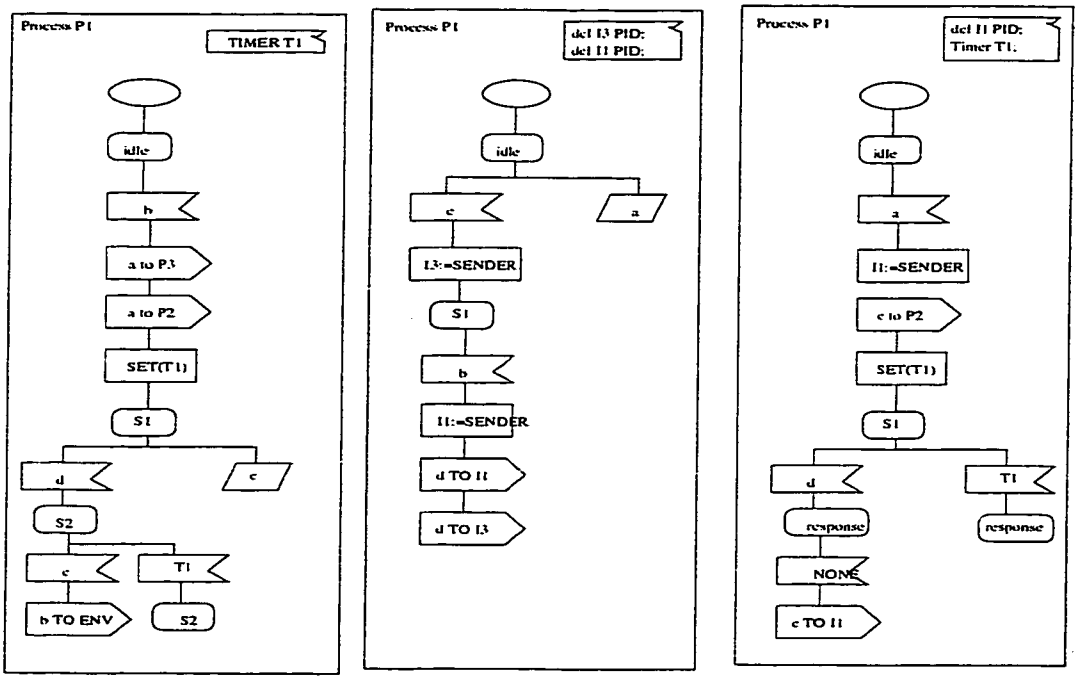

Figure 71: Example of SDL Behavior

language statements into SDL language statements, one notation in TROM language may require several notations in the SDL language. Table 14 shows the mapping between the two languages. However, we added here the mapping of Parameter-Specification in TROM into SDL. Each entry of the table consists of a TROM notation followed by its corresponding one in SDL.

Several assumptions have been made in order to remain faithful to the TROM semantics and to make the translation as natural as possible. These assumptions are:

1. Time constraints in TROM are simulated using timers and duration constructs in SDL. If a transition is supposed to be fired between two time constraints, two timers in SDL are defined to measure the duration of time. Before the expiration of the first timer, no action happens. After its expiration, the transition should happen before the expiration of the second timer. Otherwise, an error message is issued.

2. Since TROM is based on abstract modeling, only the correct behavior is described. However, since SDL is one step closer than the abstract model to implementation, all possible behaviors including incorrect ones and exceptions are modeled.

TROM		SDL
@		Gate,Channel,Signal Route
TROM end		BLOCK PROCESS ENDPROCESS ENDBLOCK
Events		STATE ENDSTATE
Attributes		Signal, Channel, Signal Route, newtype, dcl
Attribute-Function		Included inside the STATE
Parameter-Specification		Included with the signal parameter list
Transition-Specification		Included in the STATE ENDSTATE It should be simulated
Time-Constraints		Timer, Duration, Receiving the expiration signal
SCS End		SYSTEM ENDSYSTEM
Instantiate		Should be simulated by statically generating the number of instances from the created BLOCK type
Configure		Should be simulated to define the number of channels, gates, and signal routes based on the appropriate types.

Table 14: Mapping TROM notations into SDL notations

3. Incorrect time constrained behaviors are simulated in SDL by introducing signals that indicate error conditions to the system objects.

4. In SDL, the default mode of signal transmission is asynchronous. However, since TROM event triggering is synchronous, it was necessary to make some conventions to make the SDL signal transmission synchronous. Basically, two means of synchronization are considered:

   - Rendez-vous channel: This transmission mode is used with simple types of processes or blocks but not with object types. This mode of transmission means that the sender does not send the signal as long as the receiver is not ready to receive it.
   - Wait state: one possible alternative for synchronization simulation is to add a *wait* state. This state is used for two purposes: either to make the process wait until a required condition happens or just to respect the ordering of the events in the system.

6.3 SDL Model of the ATM ABR Protocol

Based on the described structure and behavior described in Chapter 5, we present here the SDL model of the ABR Protocol.

6.3.1 ABR System

Figure 72 depicts the ABR protocol. It is represented as a system composed of a Block *SWITCH* and two instances of a block type *ENDSYSTEMS*, *RS* and *RD*.

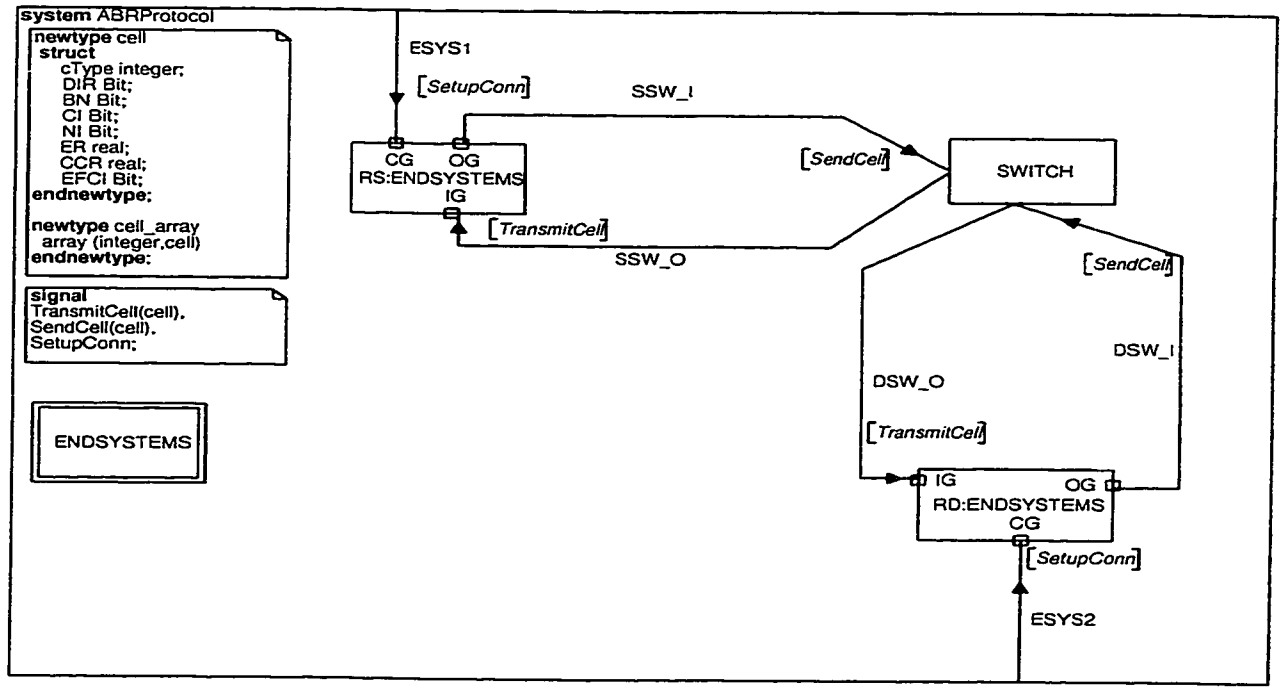

Figure 72: ABR Protocol System

The block instance *RS* communicates with the block *SWITCH* via two signal routes *SSW_I*, and *SSW_O*. The block instance *RD* communicates with the block *SWITCH* via two signal routes *DSW_I*, and *DSW_O*. Each instance of the block type *ENDSYSTEMS* has three gates to communicate: *IG* where it receives the input signal *TransmitCell* from the *SWITCH* block, *OG* where it sends the output signal *SendCell* to the *SWITCH* block, and *CG* where it receives the input signal *SetupConn* from the environment. The system has two new declared types. The first is the type *cell* that represent the parameters associated with a cell. It is a structure composed of the fields: *cType*, an integer designating the type of the cell (FRM, BRM, or data

cell), *CI, NI, DIR, BN, EFCI* of type bit, *ER*, and *CCR* of type real. The second is *cell_array*, an array where cells are to be stored. The signals *TransmitCell* and *SendCell* communicated among the blocks are parameterized signals with the parameter type *cell*.

6.3.2 End System Block Type

In Figure 73, we show the structure of the *ENDSYSTEMS* block type. It is composed of 5 processes: *DESTINATION, QUEUE, SOURCE, SQUEUE,* and *SCHEDULER*.

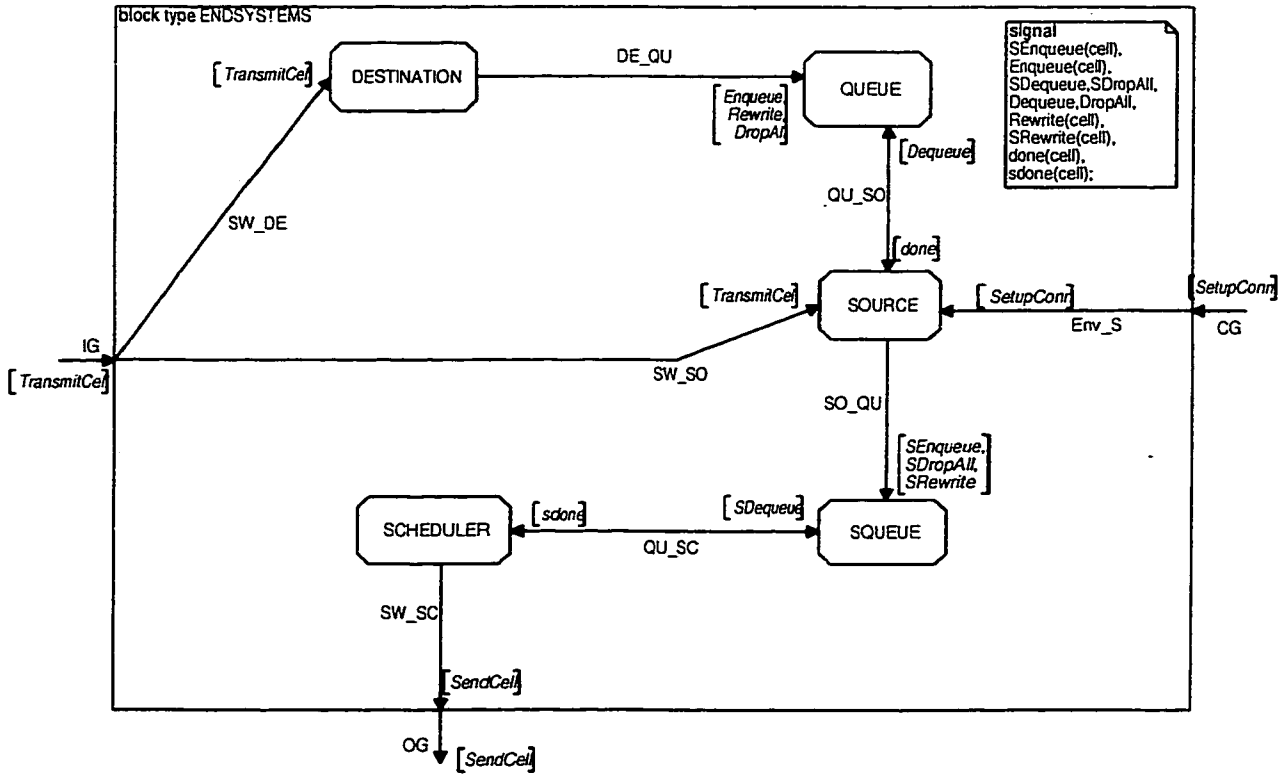

Figure 73: End System Block Type

The *ENDSYSTEMS* block receives the input signal *TransmitCell* via the gate *IG* and through the channels *SW_DE* and *SW_SO*. This signal can be received by the *DESTINATION* and *SOURCE* processes. The *ENDSYSTEMS* receives also the input signal *SetupConn* from the environment through the gate *CG*. This signal can be received by the *SOURCE* process via the channel *Env_S*. The *DESTINATION* process communicates with the *QUEUE* process via channel *DE_QU* by sending the output signals *Enqueue, DropAll,* and *Rewrite*. The *QUEUE* process communicates with

the *SOURCE* process via the channel *QU_SO* by receiving the input signal *Dequeue* and sending the signal *done*. The *SOURCE* process communicates with the process *SQUEUE* via the channel *SO_QU* by sending the signals *SEnqueue, SDropAll,* and *SRewrite*. The *SQUEUE* process communicates with the *SCHEDULER* process via the channel *SQ_SC* by receiving the input signal *SDequeue* and sending the signal *sdone*. The *SCHEDULER* process communicates with the *SWITCH* block via the channel *SW_SC* connected to the gate *OGS99*. This gate connects the outside signal route between *ENSYSTEMS* and *SWITCH* blocks. The *SCHEDULER* sends to the *SWITCH* the signal *SendCell*. The signals *Enqueue, SEnqueue, done,* and *sdone* communicated among the processes are parameterized signals with the parameter of type *cell*.

Destination Process

In Figure 74, we show the *DESTINATION* process diagram.

Queue Process

In Figure 75, we show the *QUEUE* process diagram.

Source Process

In Figure 76 and 77, we show the *SOURCE* process diagram.

SQueue Process

In Figure 78, we show the *SQUEUE* process diagram.

Scheduler Process

In Figures 79 and 80, we show the *SCHEDULER* process diagram.

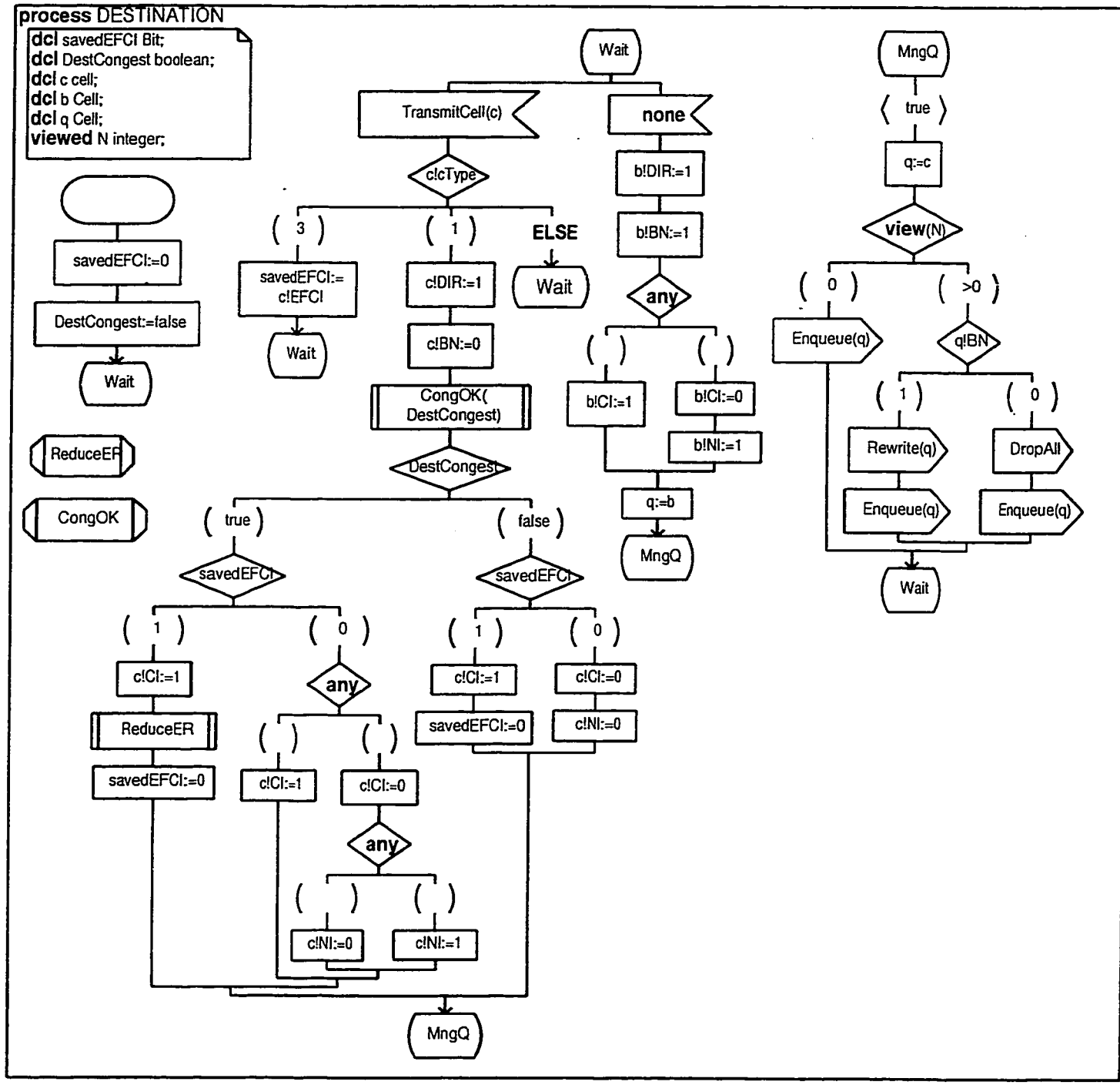

Figure 74: Destination Process

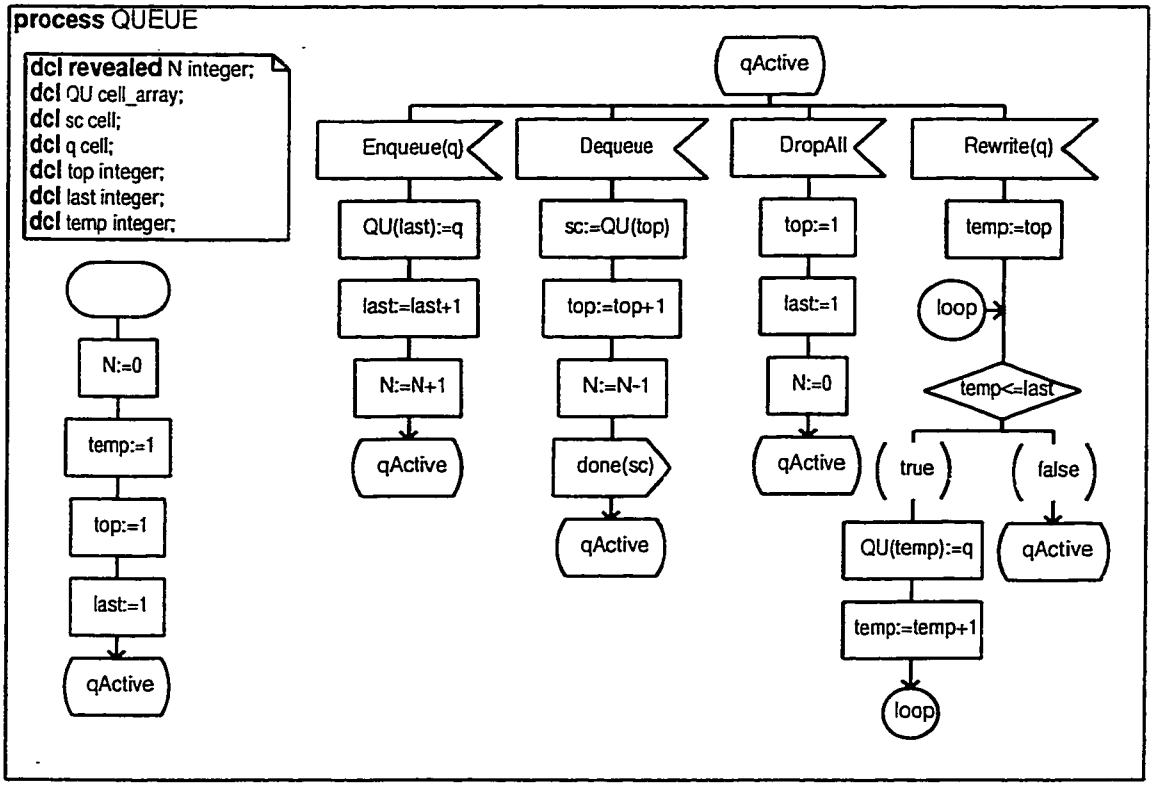

Figure 75: Queue Process

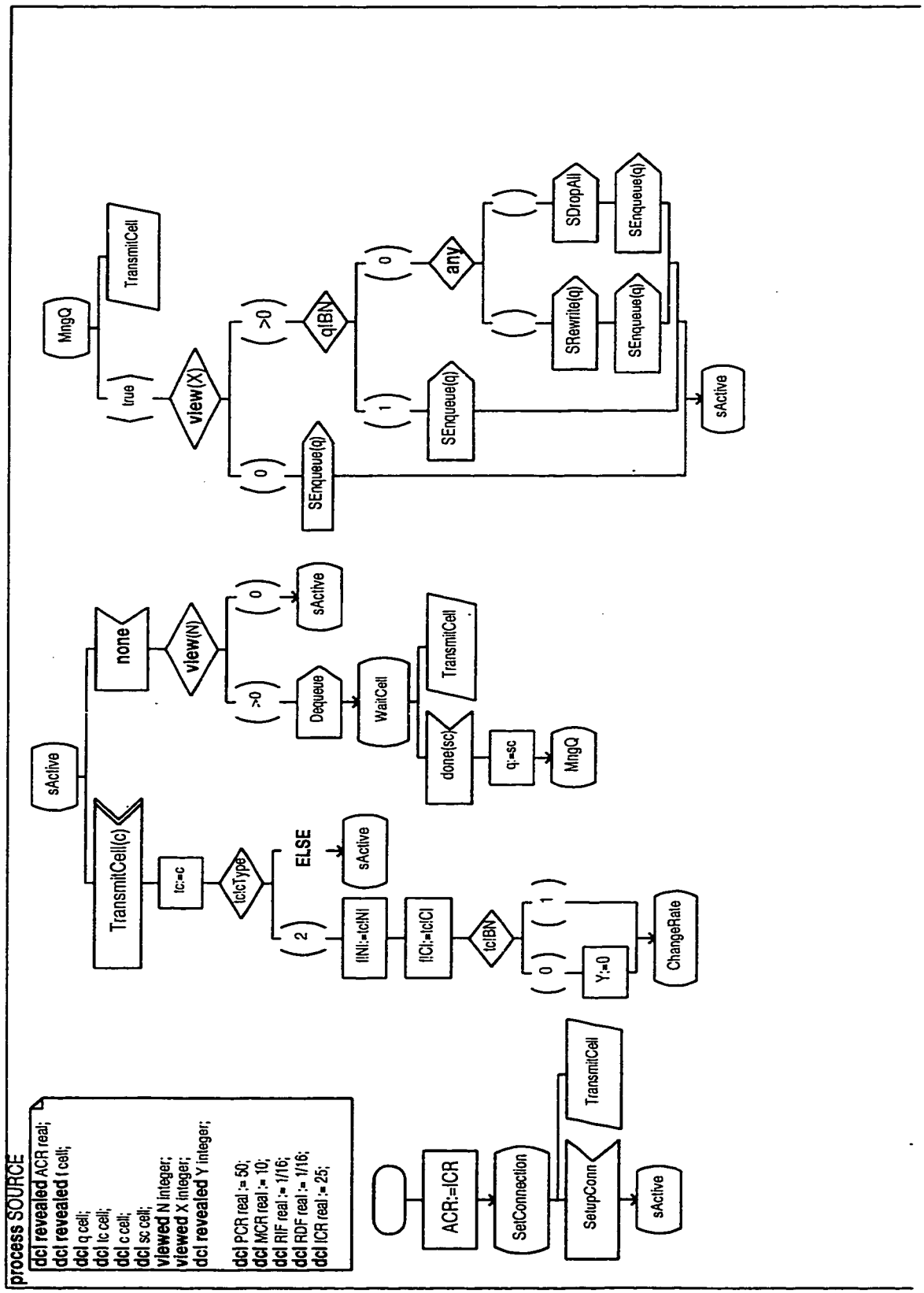

Figure 76: Source Process

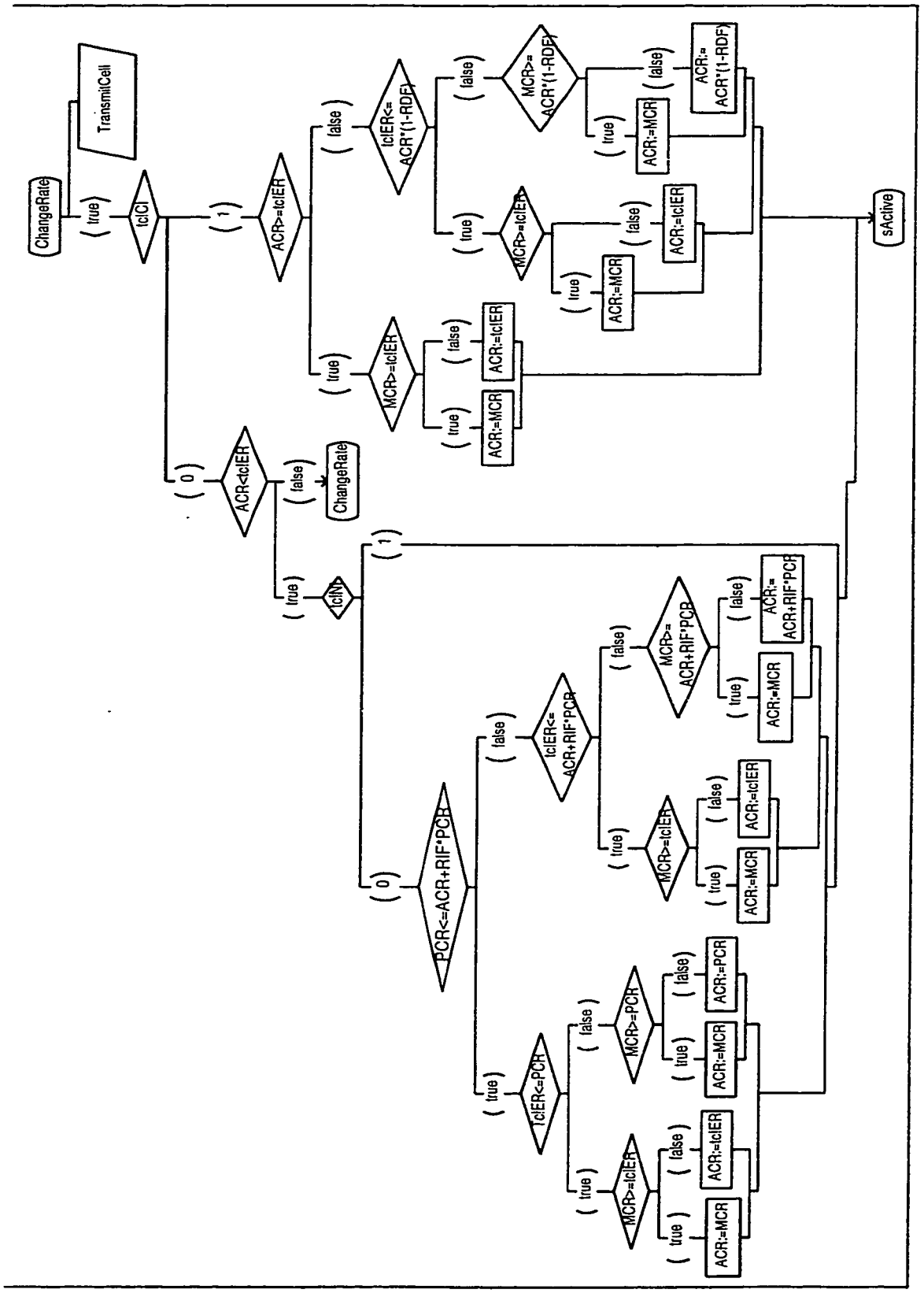

Figure 77: Source Process - Continued

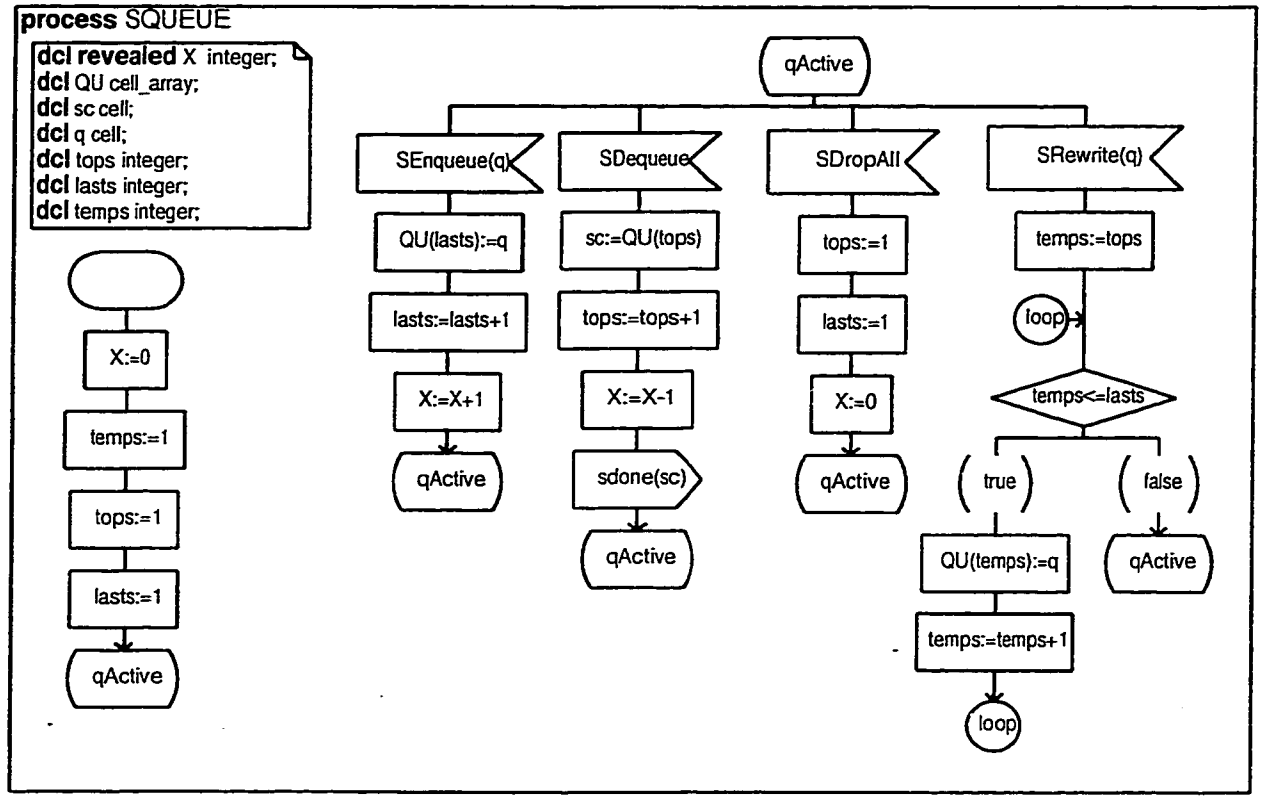

Figure 78: SQueue Process

6.3.3 Switch Block

In Figure 81, we show the structure of the *SWITCH* block. It is composed of two instances of process type *RECEIVER* namely *SR* and *DR*, two instances of process type *SWQUEUE* namely *SQ* and *DQ*, and two instances of process type *SSENDER* namely *SS* and *DS*.

The process type *RECEIVER* has two gates. The first gate, *SWI*, is used to connect the *RECEIVER* instances to the outside channels of the *SWITCH*, *SSW_I* and *DSW_I*. The second gate, *QR*, is used to connect the channel where the communication with the *SWQUEUE* process type occurs. The process type *SENDER* has two gates. The first gate, *SWO*, is used to connect the *SENDER* instances to the outside channels of the *SWITCH*, *SSW_O* and *DSW_O*. The second, *QS*, is used to connect the channel where the communication with the *SWQUEUE* process type occurs. The process type *SWQUEUE* has two gates. The first is *RQ* used to connect the channel where the communication with the *RECEIVER* process type occurs. The second is *SQ* used to connect the channel where the communication with the *SSENDER* process type

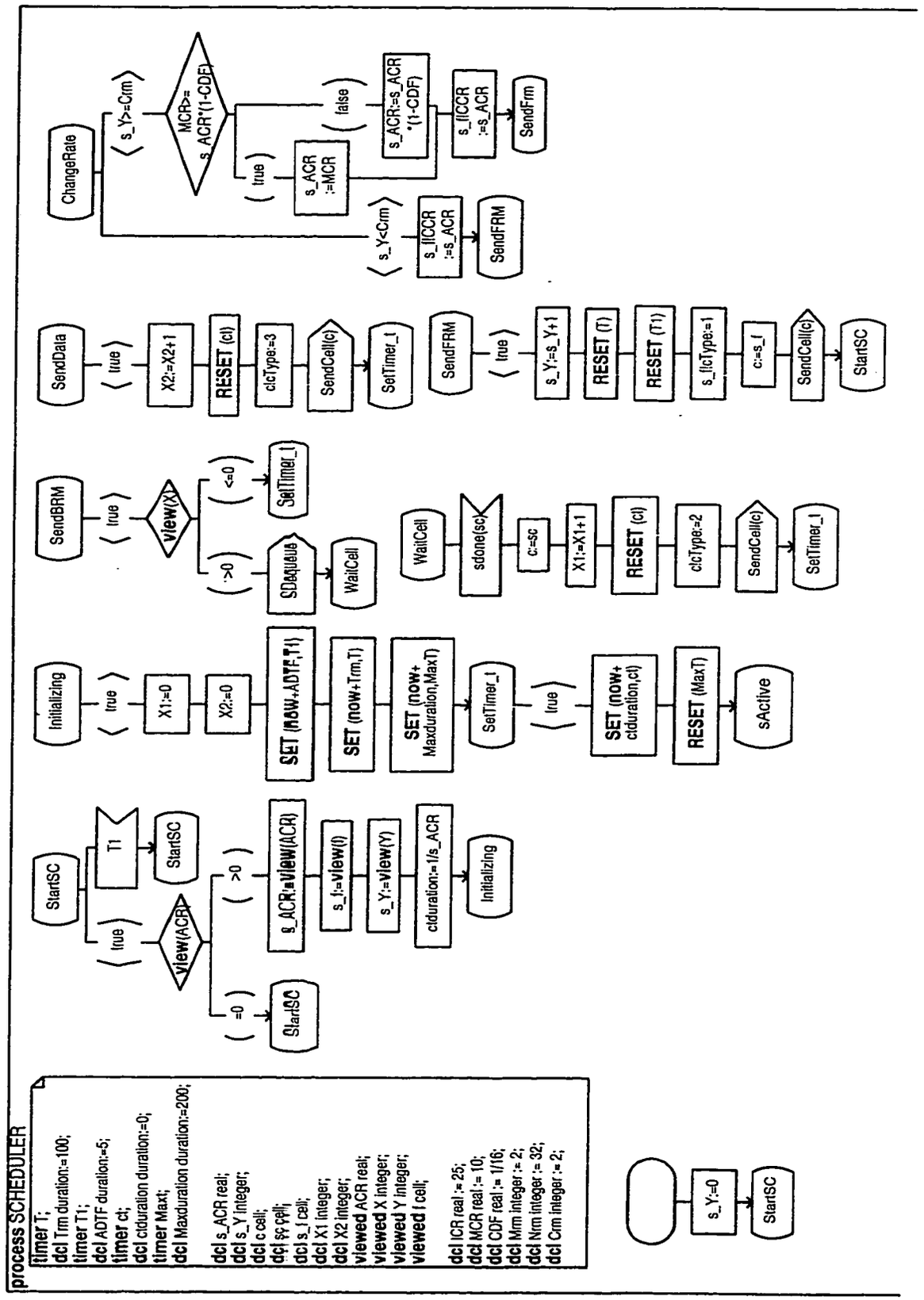

Figure 79: Scheduler Process

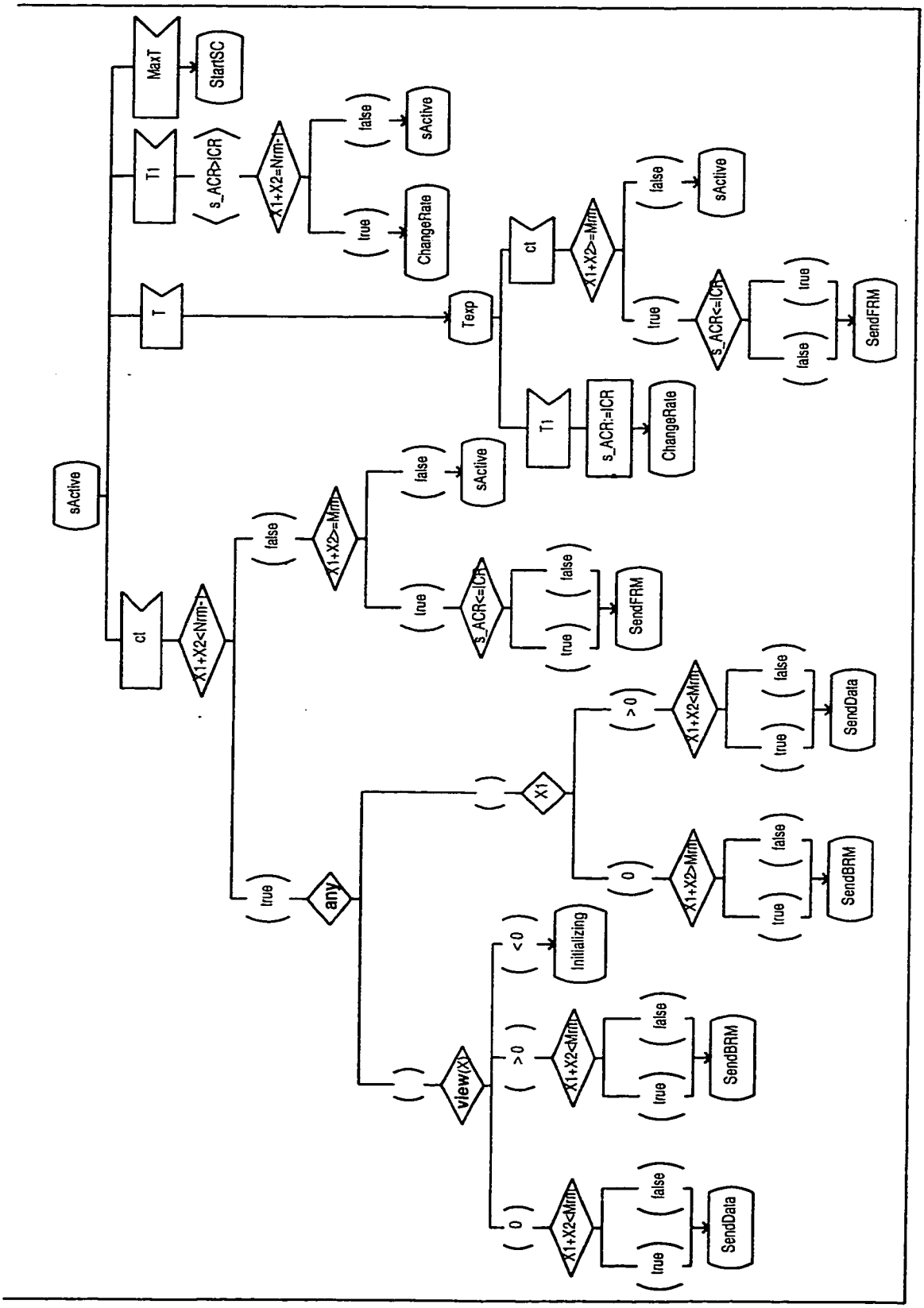

Figure 80: Scheduler Process - Continued

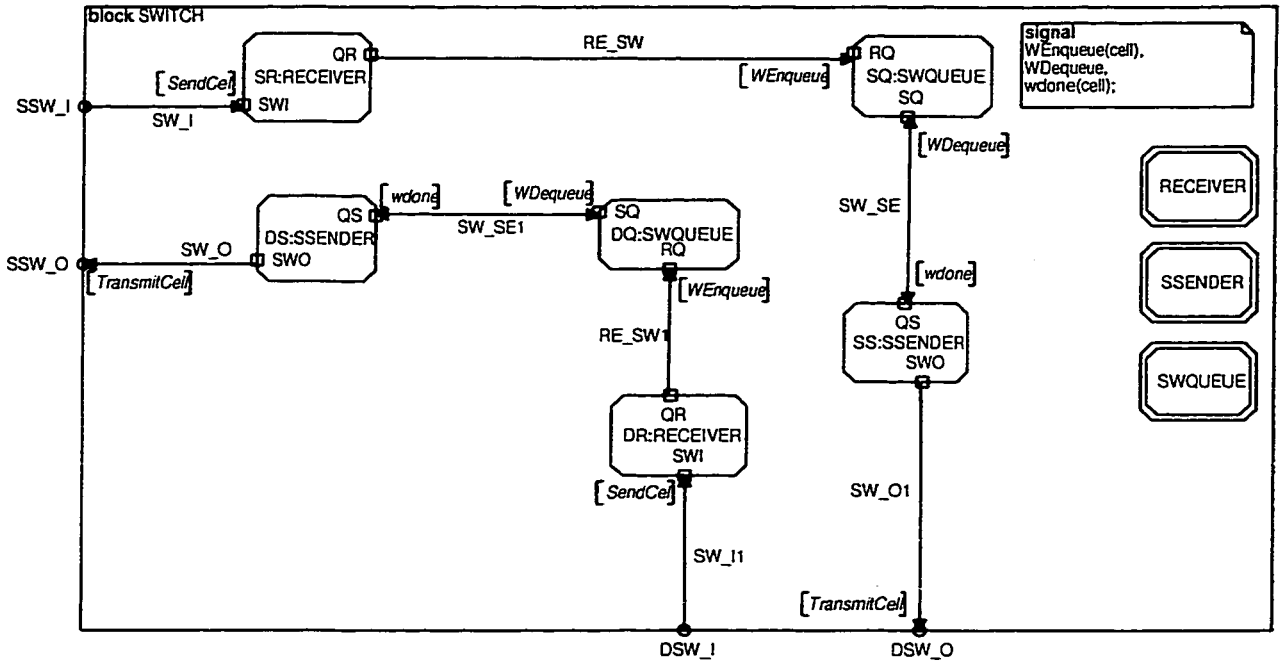

Figure 81: Switch Block

occurs.

The *RECEIVER* process instances *SR* and *DR* communicate with the *QUEUE* process instances *SQ* and *RQ* via channels *RE_SW* and *RE_SW1* respectively by sending the output signals *Enqueue, DropAll,* and *Rewrite.* They communicate with the outside blocks by receiving the input signal *SendCell* via channels *SW_I* and *SW_I1* respectively. The *SWQUEUE* process instances *SQ* and *DQ* communicate with the *RECEIVER* process instances *SR* and *DR* via channels *RE_SW* and *RE_SW1* respectively by receiving the input signals *Enqueue, DropAll,* and *Rewrite.* They communicates also with the *SSENDER* process instances *SS* and *DS* via channels *SW_SE* and *SW_SE1* respectively by sending the output signal *wdone* and receiving the input signal *WDequeue.* The *SSENDER* process instances *SR* and *DR* communicate with the *QUEUE* process instances *SQ* and *RQ* via channels *RE_SW* and *RE_SW1* respectively by sending the output signals *Enqueue, DropAll,* and *Rewrite.* They communicate with the outside blocks by sending the output signal *TransmitCell* via channels *SW_O* and *SW_O1* respectively.

The signals *WEnqueue, WDequeue,* and *wdone* communicated among the processes are parameterized signals with the parameter of type *cell.*

Receiver Process Type

In Figure 82, we show the *RECEIVER* process type diagram.

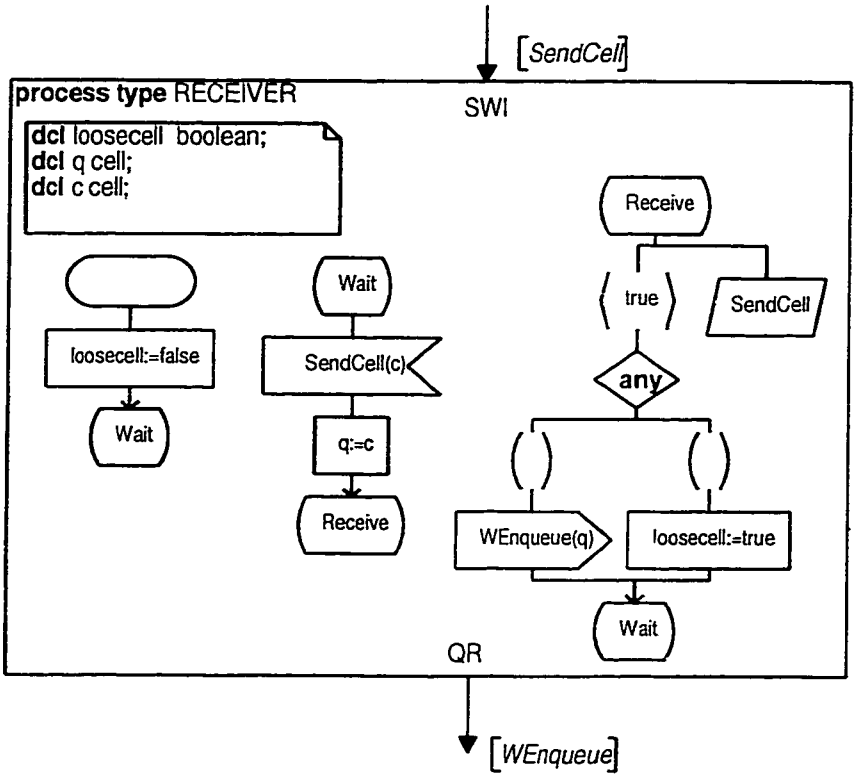

Figure 82: Receiver Process

SWQueue Process Type

In Figure 83, we show the *SWQUEUE* process type diagram.

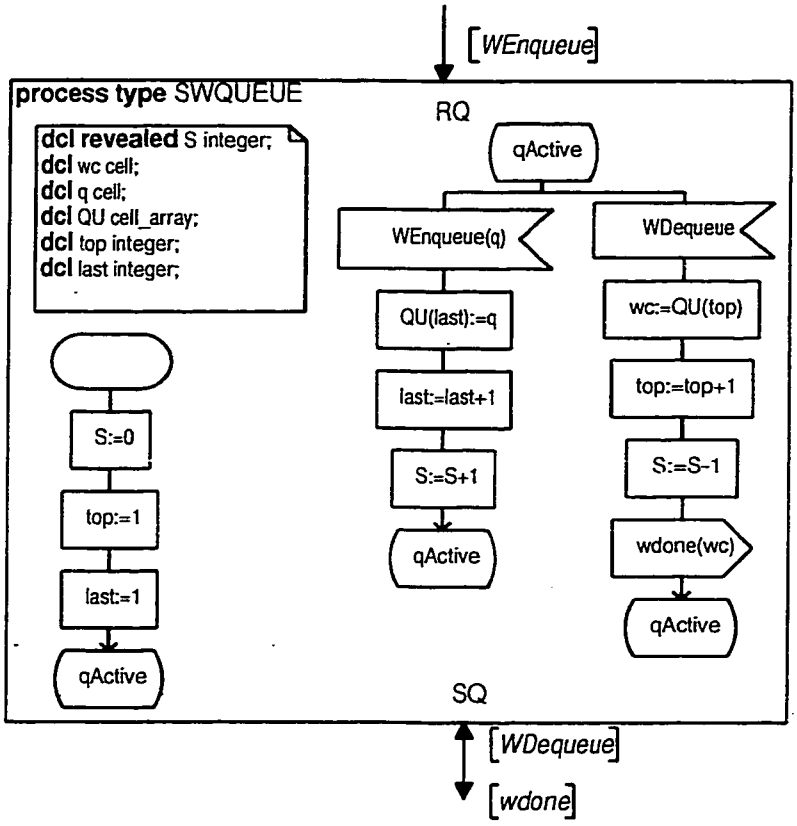

Figure 83: SWQueue Process

Sender Process Type

In Figure 84, we show the *SENDER* process type diagram.

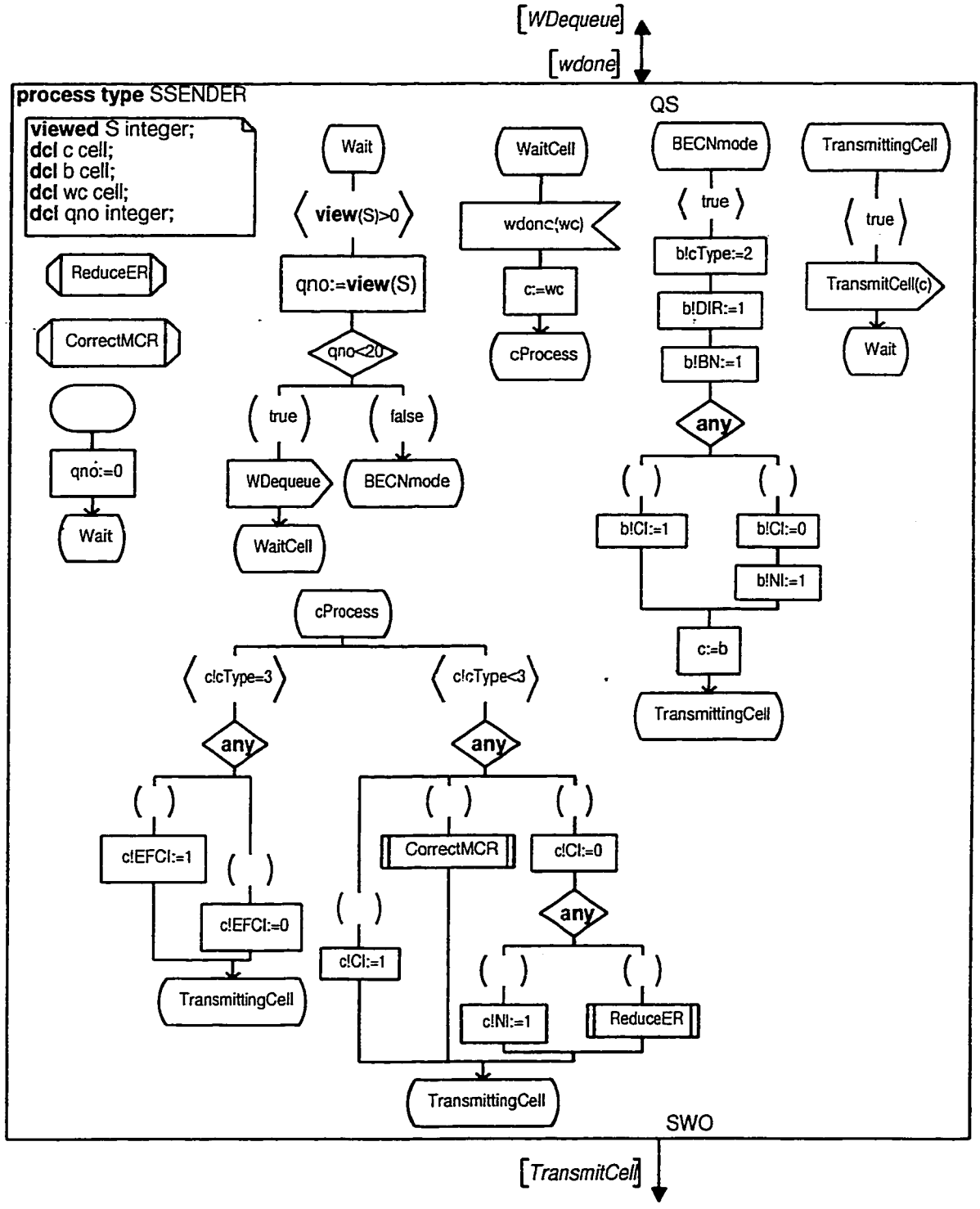

Figure 84: Sender Process

6.4 Verification and Validation

Verification is to prove that a system is correctly built by detecting design errors. Validation is to prove that the system behavior fulfills its expected behavior according to the requirements [OGS99].
We present here the facilities provided by ObjectGEODE to perform the analysis of an SDL system. We then check for several verification and validation properties in the ABR Protocol system modeled using SDL.

6.4.1 System Analysis in ObjectGEODE

The analysis in OG is based on simulation. After the specification of a system is checked for syntax and semantics errors, simulating the described system can be done in one of three modes:

- Random simulation. The tool chooses one possible execution of the system and performs it. The outcome is a trace that reflects a full execution of the system.
- Exhaustive simulation. The tool performs all the possible executions of the system and builds a state graph that reflects the state space of the system executions.
- Interactive simulation. The user controls the flow of the simulation. He chooses the transitions to fire and executes them. Again, the outcome would be in the form of a trace of fired transitions.

In addition to simulation, the ObjectGEODE tool-set includes a model checker that allows checking a system's behavior with respect to a specified property. The model checker builds a state graph representing the system and the property to be checked. The graph is then traversed in a reachability analysis to check whether the property holds in any or all the paths of the graph [OGS99]. The properties that can be checked in an SDL model usually fit into one of the following two categories:

- Verification: These are general properties that can be automatically detected through verification. They include: deadlocks, livelocks, parts of the model that are never executed, queue overflow; and dynamic errors (exceptions) such as type overflows, limits of an array exceeded, and illegal outputs [OGS99].

- Validation: Such properties are usually specific to a system's model. They can be described in ObjectGEODE using one of the following three formal representations:

  1. MSC (Message Sequence Chart) observers. They describe the signal sequences that the specification must respect [OGS99].
  2. Stop conditions. They are a kind of invariants to specify properties of the SDL model [OGS99].
  3. GOAL (Geode Observation Automata Language) observers. They are used to specify any property the SDL system must respect. They are used to specify properties impossible to describe using MSCs or stop conditions [OGS99].

6.4.2 ABR Protocol Analysis

In order to analyze the SDL model of the ABR Protocol, we use stop conditions and GOAL observers. Stop conditions are Boolean expressions (assertions) evaluated after every simulation step. GOAL observers usually express patterns to be detected in the execution of our system. The GOAL language permits to specify properties as SDL processes with some syntactical differences. The processes corresponding to properties are called observers and they are usually described in terms of entities (objects, signals...) of the tested system. An observer execution is usually labeled as success or error based on the state it leads to. The checking proceeds the same as with the simulation. However, this time, the simulation tool carries out the synchronous product of the observer and the specification of the system. This amounts to reducing the state graph to be searched for executions leading to success states and executions leading to error states. However, since our system model is large (in terms of states, variables, and transitions), the synchronous product of the system and the observers is relatively large with respect to the available resources, namely the system memory. Therefore, with the observers, it was not feasible to check our system properties using the exhaustive simulation. We used the exhaustive mode of simulation to verify the system against general properties. On the other hand, we checked the system properties using the random mode simulation since it does not require building a complete state graph.

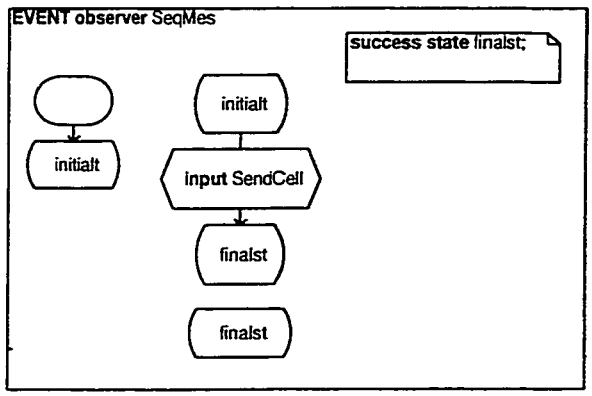

Figure 85: SendCell Input Signal Observer

In the case of the ABR model, we check and verify that the system respects the following properties:

- General properties: deadlocks, livelocks, exceptions, and unsuccessful loops. These are checked by running the exhaustive simulation. The simulator reports whether any of the mentioned problems is met during the simulation.

- *Safety properties*: they represent the guarantee that nothing bad happens or the avoidance of illegal incidents in general; e.g., illegal inputs [Hal00]. For example, we checked the signal loss property. The following stop conditions represent the property for input and output signals respectively:
  false input and *false output*

- *Liveness properties*: it requires that allowed events not be postponed forever [Hal00] and that the model's behavior will always eventually become an expected behavior. In our case, we check that every signal is eventually delivered. As an illustration, the observer in Figure 85 makes sure that the input signal *SendCell* is eventually received. When *SendCell* is received, the observer goes to a success state.

- *Message Order*: here we check if the order of delivered and received messages is respected in the system. for example we make sure that in the *SWITCH* block, the message *WEnqueue* is never delivered before the reception of *SendCell* message. The following stop conditions express this property:
  true input SendCell and *true output WEnqueue*

Chapter 7

Conclusion and Future Work

In this thesis, we introduced parameterized events to the TROM formalism. Our motivation has been to address the state explosion problem in TROM while adding more clarity to the system model and enhancing the expressiveness of the formalism. We defined formal syntax and semantics for parameterized events. In order to accommodate the event parameterization into the TROMLAB environment, we extended the TROM grammar to include the formal representation of parameters. In addition, we augmented the Real-Time UML notation to be able to visually model the event parameterization, and reengineered some of the TROMLAB tools to acquire this parameterization. We modified the GRC–Translator to automatically generate the formal representation of parameterized events out of the Rose models. We also modified the Interpreter to parse, syntactically check the produced formal representations, and construct the internal representation of parameters for simulation purposes.

We illustrated our approach by remodeling the Train–Gate–Controller system for parameterized events. This is a benchmark example that has been studied by real-time reactive systems community.
Parameterizing the events of Train–Gate–Controller model enhanced the clarity and expressiveness of the model.

As a case study, we reworked an example from [LDU96], which considers deriving a formal specification for the ATM ABR protocol source and destination from english

specification. However, our model considers, in addition to the end system description, the behavior of the ATM ABR switch. We used the modified Translator and Interpreter to produce the formal specification of the model and syntactically check the produced specification, respectively. Finally, we mapped the TROM model of our case study into SDL, and we verified its correctness using the ObjectGEODE tool set. We checked the model for safety and liveness properties.

However, our work did not include upgrading the simulator of the TROMLAB since the upgrades of this tool are not trivial. In addition, the whole design of the simulator needs to be thoroughly reconsidered. Also, the Verification Assistant (PVS) need to be considered for modifications to acquire parameterized events. Therefore, these tasks are left for future consideration.
Also, one of the main issues that can be considered for future work is extending the Rational Rose tool to model composite classes given the existing semantics.

Bibliography

[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.

[ATM96] ATM Forum. *ATM Traffic Management Specification Version 4.0*, April 1996.

[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.

[Hal00] H. Hallal. Divergence-free supervisory control and applications. Master's thesis, Department of Electrical and Computer Engineering, McGill University, Montréal, Canada, July 2000.

[HRS98] Huber M.N. Handel R. and Schroder S. *ATM Networks: Concepts, Protocols, Applications.* Addison Wesley Longman Ltd., 1998.

[JR96] Fahmy S. Goyal R. Jain R., Kalyanaraman S. Source behavior for atm abr traffic management: An explanation. *IEEE Communications Magazine*, November 1996.

[LDU96] Moh W. M. Lee D., Ramakrishnan K. K. and Shankar A. U. Protocol specification using parameterized communicating extended finite state machines - a case study of the atm abr rate control scheme. In *Proceedings of International Conference on Network Protocols, ICNP*, October 1996.

[MA98a] D. Muthiayen and V. S. Alagar. Formalizing UML for rigorous software development. In *Proceedings of Workshop on Formalizing UML. Why? How? held at Thirteenth ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA98*, Vancouver, Canada, October 1998.

[MA98b] D. Muthiayen and V. S. Alagar. A UML-based methodology for real-time reactive system development. In *Proceedings of Workshop on Behavioral Semantics of Object-Oriented Business and System Specifications held at Thirteenth ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA98*, Vancouver, Canada, October 1998.

[MD00] Khendek F. Sefidcon A. Muthiayen D., Alagar V.S. An approach to a synthesis of formal and visual description techniques for the development of real-time reactive systems. In *Proceedings of Real-Time Computer System and Applications*, Cheju, South Korea, December 2000.

[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.

[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.

[OGS99] VERILOG. *ObjectGEODE SDL Simulator Reference Manual*, April 1999.

[Pom99] F. Pompeo. A formal verification assistant for TROMLAB environment. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, March 1999.

[Pop99] O. Popistas. Rose-grc translator: Mapping UML visual models onto formal specifications. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, April 1999.

[Rat98] Rational Software Corporation. *Rational Rose 98 Enterprise Edition Rose Extensibility Interface*, February 1998.

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

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

[TLL99] Telelogic. *Specification and Description Language (SDL)*, 1999. Available through www.telelogic.com.

[Zha00] L. Zhang. Automatic code generation for real-time reactive systems in TROMLAB environment. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, August 2000.

Appendix A

Parameters Data Structures in the Rose-GRC Translator

RoseScript supports user-defined types. We declare here the parameters types added to the translator's internal structure.

```
Type GRCParameter
    eventName As String          'the event that can have the parameter list
    StartState As String         'the start state of the transition
    Parms(MaxParms) As String ' array of parameters appearing
                                 'with the event
                                 'certain transition
    NumParms As Integer          'number of parameters of the parameter
                                 'list of the event
End Type

Type GRCParmEvent
    eventName As String          ' Event name
    ParmArray(MaxParameters)  ' Array of all possible parameters
                                 that appear with the event
    NumParameters As Integer     number of parameters associated with the event
End Type
```

the GRCTransition and GRC types is changed as follows:

```
Type GRCTransition
    Trans As Transition          ' Rose transition
    RNum As String               ' Internal title for the transition, of format
                                 ' R#, where # is a natural number >0
    EventNum As Integer          ' position of transition's triggering event in
                                 ' array GRCEvents
    PortCondition As String      ' String extracted from Rose transition's
                                 ' Action field
    EnablingCondition As String ' String extracted from Rose transition's
```

```
                                ' guard condition
    PostCondition As String     ' String extracted from Rose transition's
                                ' guard condition.
    ParameterList As String     ' String extracted from Rose transition's
                                ' arguments' where parameters and their values are specified
End Type

Type GRC
    GRCClass As Class           ' Rose class with stereotype GRC
    PortTypes(MaxPortTypes) As GRCPortType
                                'array of port types
    NumPortTypes As Integer     ' number of valid port types in array PortTypes
    GRCEvents(MaxEvents) As GRCEvent
                                ' array of events
    NumEvents As Integer        ' number of valid events in array GRCEvents
    GRCAttributes(MaxAttributes) As GRCAttribute
                                ' array of attributes
    LSLTraits(MaxTraits) As String
                                ' array of LSL traits
    NumLSLTraits As Integer     ' number of valid LSL traits in array LSLTraits
    GRCStates(MaxStates) As GRCState
                                ' array of states
    NumStates As Integer        ' number of valid states in array GRCStates
    GRCParmEvents(MaxEvents) As GRCParmEvent
                                ' array of parameters
                                ' associated with states and events
    NumParmEvents As Integer    ' number of valid parameterized events
    GRCTransitions(MaxTransitions) As GRCTransition
                                ' array of transition specifications
    NumTransitions As Integer   ' number of valid transitions in array
                                ' GRCTransitions
    GRCTimeConstraints(MaxConstraints) As GRCTimeConstraint
                                ' array of time constraints
    NumTimeConstraints As Integer
```

' number of valid time constraints in array
' GRCTimeConstraints

End Type

Algorithms Implementing Parameters in the Rose-GRC Translator

We describe the algorithms necessary to extract the parameters from the Rose model. This requires modifying some of the existing algorithms and adding new ones. The functions implementing these algorithms are executed only after a parameter flag is set on. This flag is set by the user from the GUI of the translator.
The algorithm for extracting attributes is slightly changed to extract parameters and include them in the attribute section of the GRC internal structure.

Algorithm for extracting attributes

For all attributes of the GRC class
  If stereotype is PortType then
    If type of attribute is not the name of a port type class stored..
    ..in the list of port types for the GRC then Error.
  If stereotype is DataType or Parameter then
    If attribute type is Integer or Boolean then store attribute name and..
    .. type in list of attributes for the GRC.
    If attribute type is of the form LSLtrait[par...traitType] then
      Store attribute name in list of attributes for the GRC.
      Store traitType as the type of attribute in list of attributes for the GRC.
      Store LSLtrait in the list of LSL traits for the GRC.
  else Error.
End for

The algorithm for extracting transition specifications is also changed to extract the parameter list of the triggering events.

Algorithm for extracting transition specifications

For all transitions in the state machine of the GRC class

If transition is outgoing of Rose's initial state or incoming in Rose's end state..

.. then skip.

If no trigger event specified for the transition then Error.

Store transition name, source state and target state in transition list for the GRC.

If guard condition is empty then Default port-condition and enabling-condition ..

.. to true in the transition list for the GRC.

If guard condition is not of the form port-condition && enabling-condition && ..

..time-condition then Error.

Extract port-condition and enabling-condition from guard condition.

Store port-condition and enabling-condition in list of transitions for the GRC

If trigger action is empty then default post-condition to true in transitions list..

.. for the GRC.

If trigger action is not of the form post-condition && Time-initialization ..

.. then Error.

Extract post-condition from trigger action.

Store post-condition in the list of transitions for the GRC.

if parameter flag is set on then

store the event name and the source state name in a temporary list

for each parameter entry of the event argument list

store the parameter entry in the event corresponding parameter list

End for End for

Algorithm for extracting parameter specifications

For each entry of the parameter list extracted from the transition specifications

Extract the parameter name

If parameter name is followed by the = sign and not followed by a value..

.. then Error.

If parameter name is not found in the attribute list then Error

End for

Sort parameter list by event in ascending order

Sort sorted parameter list by state in ascending order

For each entry in the list compare each two consecutive entries
if the event and state names are the same then
if corresponding parameter names are not the same
then Error.
Go to the next entry
End for
For each event in the list
For each parameter corresponding to the event
if parameter is not in attribute function of the source state of the event
then add parameter to the source state attribute function
End for
End for
For each event in the list
Store the event with all the parameters appearing with this event in the list of..
.. parameters for the GRC.
End for

Pseudo Code Implementing Parameters in the Interpreter

As the original parsers, the modified parsers are written using the *JavaCC* language (a precompiler of *Java*. We present below the pseudo code for parsing parameter specifications section and the modified code for parsing transition specifications section.

Pseudo code for parsing parameter specifications

```
  private static String getparameterlist(eventlist el,
attributelist al, PrintStream psout) throws ParseException
  {
scan through the specification
    getIdentifier;
    if identifier = "Parameter-Specifications"
    {
```

```
  get a colon and a newline characters
  get parameterlist(el,al,psout);
}
if error then exit the parser
return;
    }

   boolean get_parameterlist(eventlist evl, attributelist al,
      PrintStream psout) :

Loop until all the parameter specification lines are parsed
   ( get identifier
if identifier = "Transition-Specifications"
return true;
else
  pl =define a  new Parmlist();
  if identifier is not a valid event name in the event list
produce error
return false;
   get colon
   get identifier
   if (the identifier is not a valid attribute in the attribute list)
          or (the identifier is already a parameter for the current event)
produce error
return false;
else
add the identifier in the temporary parameter list;
   Loop until NewLine is found
      ( get comma
      get identifier
if (identifier is not a defined attribute in the attribute list)
     or (identifier is already an existing parameter in the parameter list)
produce errro
```

```
return false;
else
add the parameter in the temporary parameter list;
   add the parameters of the temporary parmlist to the event's parameter list
 return true;
```

Pseudo code for parsing transition specifications Here, we present the pseudocode for parsing the parameters in the TransitionSpecification section. Note that we do not provide all the pseudocode of the function parsing the TransitionSpecification section.

```
   trans_speclist get_transpeclist(eventlist el,
      statelist sl, attributelist al, PrintStream psout) :

   Loop through all the transitions of the transition-specification section
   get identifier
      if identifier = "Time-Constraints"
    return transition specification list;
      else
if transition specification list contains the current on
           produce error
     return null;
else
   set the label of the transition;
   get colon; get left arrow bracket;
   get identifier;
      if identifier is a valid state name then
         set the source state of thetransition;
      else
   produce error;
   return null;
   get comma;
   get identifier;
       if identifier is a valid state name then
```

```
        set the destination state of the transition;
      else
   produce error;
   return null;
   get right arrow bracket; get semicolon;
   get identifier;
if identifier is a valid event name in the event list then
   set the triggering event for the transition;
      else
   produce error;
   return null;
   if the next identifier is a left square bracket then
   {  get left square bracket;
      get identifier;
      parm_and_value = new List();
      if the identifier is a valid parameter of the triggering event
        parameter list then
   get the identifier as the parameter;
      else
   produce error;
   return null;
     if the next identifier is an equal operator then
       get the equal operator;
       get identifier;
       get the identifier as the value assigned to the parameter;
       if the parameter extracted is already in use then
         produce error;
         return null;
       else
         append the parameter and its value if any to the transition;
      if the next identifier is a comma then
        Loop until right square bracket found
          get comma;
```

```
        get identifier;
        parm_and_value = new List();
        if the identifier is a valid parameter of the triggering event
           parameter list then
    get the identifier as the parameter;
        else
    produce error;
    return null;
 if the next identifier is an equal operator then
            get the equal operator;
            get identifier;
            get the identifier as the value assigned to the parameter;
        if the parameter extracted is already in use then
            produce error;
            return null;
        else
            append the parameter and its value if any to the transition;
get right square bracket
assign the whole parameter list with their values to the transition;

.............
```

Appendix B

ABR English Specifications

This appendix provides the precise source, destination, and switch behavior verbatim from the ATM ABR specification [ATM]. All table, section, and other references in this appendix refer to those in the ATM specification.

Source Behavior The following items define the source behavior for CLP$\bar{0}$ and CLP$\bar{1}$ cell streams of a connection. By convention, the CLP$\bar{0}$ stream is referred to as inrate, and the CLP$\bar{1}$ stream is referred to as outofrate. Data cells shall not be sent with CLP$\bar{1}$.

1. The value of ACR shall never exceed PCR, nor shall it ever be less than MCR. The source shall never send in-rate cells at a rate exceeding ACR. The source may always send inrate cells at a rate less than or equal to ACR.

2. Before a source sends the first cell after connection setup, it shall set ACR to at most ICR. The first inrate cell shall be a forward RMcell.

3. After the first inrate forward RMcell, inrate cells shall be sent in the following order:
   a) The next inrate cell shall be a forward RM cell if and only if, since the last inrate forward RMcell was sent, either: i) at least Mrm inrate cells have been sent and at least Trm time has elapsed, or ii) Nrm 1 inrate cells have been sent.
   b) The next inrate cell shall be a backward RMcell if condition (a) above is not met, if a backward RM cell is waiting for transmission, and if either: i) no inrate backward RMcell has been sent since the last inrate forward RM cell, or ii) no data cell is waiting for transmission.
   c) The next inrate cell sent shall be a data cell if neither condition (a) nor condition (b) is met, and if a data cell is waiting for transmission.

4. Cells sent in accordance with source behaviors #1,#2, and #3 shall have CLP$\bar{0}$.

5. Before sending a forward inrate RM cell, if ACR>ICR and the time T that has elapsed since the last inrate forward RMcell was sent is greater than ADTF, then ACR shall be reduced to ICR.

6. Before sending an inrate forward RM cell, and following behavior #5 above, if at least CRM inrate forward RMcells have been sent since the last backward RMcell with $\overline{BN0}$ was received, then ACR shall be reduced by at least ACR × CDF, unless that reduction would result in a rate below MCR, in which case ACR shall be set to MCR.

7. After following behaviors #5 and #6 above, the ACR value shall be placed in the CCR field of the outgoing forward RMcell, but only inrate cells sent after the outgoing forward RMcell need to follow the new rate.

8. When a backward RMcell (inrate or outofrate) is received with $\overline{CI}$, then ACR shall be reduced by at least ACRRDF, unless that reduction would result in a rate below MCR, in which case ACR shall be set to MCR. If the backward RMcell has both $\overline{CI0}$ and $\overline{NI0}$, then the ACR may be increased by no more than RIFPCR, to a rate not greater than PCR. If the backward RMcell has $\overline{NI}$, the ACR shall not be increased.

9. When a backward RMcell (inrate or outofrate) is received, and after ACR is adjusted according to source behavior #8, ACR is set to at most the minimum of ACR as computed in source behavior #8, and the ER field, but no lower than MCR.

10. When generating a forward RMcell, the source shall assign values to the various RMcell fields as specified for sourcegenerated cells in Table 54.

11. Forward RMcells may be sent outofrate (i.e., not conforming to the current ACR). Outofrate forward RMcells shall not be sent at a rate greater than TCR.

12. A source shall reset EFCI on every data cell it sends.

13. The source may implement a useitorloseit policy to reduce its ACR to a value which approximated the actual cell transmission rate. Useitorloseit policies are discussed in Appendix I.8.

Notes:

1. Inrate forward and backward RMcells are included in the source rate allocated to a connection.

2. The source is responsible for handling congestion within its scheduler in a fair manner. This congestion occurs when the sum of the rates to be scheduled exceeds the output rate of the scheduler. The method for handling local congestion is implementation specific.

Destination Behavior The following items define the destination behavior for CLP$\bar{0}$ and CLP$\bar{1}$ cell streams of a connection. By convention, the CLP$\bar{0}$ stream is referred to as inrate, and the CLP$\bar{1}$ stream is referred to as outofrate.

1. When a data cell is received, its EFCI indicator is saved as the EFCI state of the connection.

2. On receiving a forward RMcell, the destination shall turn around the cell to return to the source. The DIR bit in the RMcell shall be changed from "forward" to "backward", BN shall be set to zero, and CCR, MCR, ER, CI, and NI fields in the RMcell shall be unchanged except:
   a) If the saved EFCI state is set, then the destination shall set CI$\bar{1}$ in the RM cell, and the saved EFCI state shall be reset. It is preferred that this step is performed as close to the transmission time as possible;
   b) The destination (having internal congestion) may reduce ER to whatever rate it can support and/or set CI$\bar{1}$ or NI$\bar{1}$. A destination shall either set the QL and SN fields to zero, preserve these fields, or set them in accordance with ITUT Recommendation I.371-draft. The octets defined in Table 54 as reserved may be set to 6A (hexadecimal) or left unchanged. The bits defined as reserved in Table 54 for octet 7 may be set to zero or left unchanged. The remaining fields shall be set in accordance with Section 5.10.3.1 (Note that this does not preclude looping fields back from the received RM cell).

3. If a forward RMcell is received by the destination while another turnedaround RMcell (on the same connection) is scheduled for inrate transmission:
   a) It is recommended that the contents of the old cell are overwritten by contents

of the new cell;

b) It is recommended that the old cell (after possibly having been overwritten) shall be sent outofrate; alternatively the old cell may be discarded or remain scheduledfor inrate transmission;

c) It is required that the new cell be scheduled for inrate transmission.

4. Regardless of the alternatives chosen in destination behavior #3, the contents of the older cell shall not be transmitted after the contents of a newer cell have been transmitted.

5. A destination may generate a backward RMcell without having received a forward RMcell. The rate of the backward RMcells (including both inrate and outofrate) shall be limited to 10 cells/second, per connection. When a destination generated an RMcell, it shall set either CI$\bar{1}$ or NI$\bar{1}$, shall set set BN$\bar{1}$, and shall set the direction to backward. The destination shall assign values to the various RMcell fields as specified for destination generated cells in Table 54.

6. When a forward RMcell with CLP$\bar{1}$ is turned around it may be sent inrate (with CLP$\bar{0}$) or outofrate (with CLP$\bar{1}$)

Notes:

1. "Turn around" designates a destination process of transmitting a backward RMcell in response to having received a forward RMcell.

2. It is recommended to turn around as many RMcells as possible to minimize turnaround delay, first by using inrate opportunities and then by using outofrate opportunities as available. Issues regarding turning RMcells around are discussed in Appendix I.7.

Switch Behavior The following items define the switch behavior for CLP=0 and CLP=1 cell streams of a connection. By convention, the CLP=0 stream is referred to as in-rate, and the CLP=1 stream is referred to as out-of-rate. Data cells shall not be sent with CLP=1.

1. A switch shall implement at least one of the following methods to control congestion at queuing points:

   a) *EFCI marking:* The switch may set the EFCI state in the data cell headers;

b) *Relative Rate Marking:* The switch may set CI=1 or NI=1 in forward and/or backward RM-cells;

c) *Explicit Rate Marking:* The switch may reduce the ER field of forward and/or backward RM-cells (Explicit Rate Marking) ;

d) *VS/VD Control:* The switch may segment the ABR control loop using a virtual source and destination.

2. A switch may generate a backward RM-cell. The rate of these backward RM-cells (including both in-rate and out-of-rate) shall be limited to 10 cells/second, per connection. When a switch generates an RM-cell it shall set either CI=1 or NI=1, shall set BN=1, and shall set the direction to backward. The switch shall assign values to the various RM-cell fields as specified for switch-generated cells in Table 5-4.

3. RM-cells may be transmitted out of sequence with respect to data cells. Sequence integrity within the RM-cell stream must be maintained.

4. For RM-cells that transit a switch (i.e., are received and then forwarded), the values of the various fields before the CRC-10 shall be unchanged except:

   a) CI, NI, and ER may be modified as noted in #1 above

   b) RA, QL, and SN may be set in accordance with ITU-T Recommendation I.371-draft

   c) MCR may be corrected to the connection's MCR if the incoming MCR value is incorrect.

5. The switch may implement a use-it-or-lose-it policy to reduce an ACR to a value which approximates the actual cell transmission rate from the source. Use-it-or-lose-it policies are discussed in Appendix I.8.

Notes:

1. A switch queuing point is a point of resource contention where cells may be potentially delayed or lost. A switch may contain multiple queuing points.

2. Some example switch mechanisms are presented in Appendix I.5.

3. The implications of combinations of the above methods is beyond the scope of this specification.