ROSE-GRC TRANSLATOR: MAPPING UML VISUAL MODELS ONTO FORMAL SPECIFICATIONS

Oana-Mirela Popistas

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 1999

0-612-43557-1

Abstract

Rose-GRC Translator: Mapping UML Visual Models onto Formal Specifications

Oana-Mirela Popistas

Real-time reactive systems are among the most difficult systems to design because of the complex functional and timing requirements that must be satisfied. Visual models serve to break the complexity barrier. allowing the developer to comprehend and reason with graphical representations. The graphical representations by themselves are not sufficient—they are informal and lack well-defined meaning. This thesis allows the description of classes. statechart diagrams. and collaboration diagrams of reactive system components to be constructed graphically using UML notation in the Rose environment. and maps them to a formal notation, which can be subjected to a rigorous analysis. such as validation and verification, prior to committing to an implementation. The mapping is implemented by a translator using RoseScript. the language provided with the Rose Extensibility Interface of Rational Rose.

To my children: Daniel-Alexander and Christina-Andrea

Acknowledgments

A warm thank you goes to my supervisor, Dr. V. S. Alagar, who guided this work with good advice, patience and understanding, and provided good technical support. His availability was always much appreciated and he made me feel that my contribution was important in the overall project. I would like to thank Dr. J. W. Atwood who helped me decide to undertake the Masters program and who, throughout the years, always had a good word of encouragement. I was first introduced to UML by Dr. G. Butler, and I would like to thank him for his wise advice and understanding. Also, I would like to thank all Faculty and Staff and my fellow graduates for their support during these years. In particular I thank Darmalingum Muthiayen, with whom I worked closely in the last months and who was an excellent role model, both as a researcher and as a professional.

I thank Ericsson Research Canada and my managers for the continuous support that they provided for the courses, and mainly for their contribution to this thesis. The Rational Rose license facilitated the implementation of the Rose-GRC Translator.

On a personal level, I thank my husband Adrian, for his support and help. Even in his busiest times, as an employee, part-time Master student and father of two, he found the energy to encourage me. I thank my two wonderful children, Daniel and Christina who, without realizing it, gave me the motivation to complete this work. I thank my parents and parents-in-law for their continuous help throughout the years, for loving and caring for our children.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reactive systems maintain an ongoing continuous interaction with their environment through stimulus and response. For real-time reactive systems, the stimulus-response behavior is regulated by timing constraints. Such systems find application in areas such as transportation, workshop automation, mobile telephony, process control, and strategic defence systems. The behavior of a reactive system is in general *infinite*: a process in a reactive system is usually non-terminating. In this respect reactive systems are different from common transformational systems, which may be regarded as functions from input available at the start of the computation to output provided on termination. Reactive systems are also different from interactive systems, such as a human-computer interface. The major distinction is in the available synchronization mechanism: an interactive system may wait for a reply from its environment: on the contrary a reactive system such as a shutdown system in a nuclear power plant is fully responsible for synchronization with its environment. Consequently, it must satisfy two important requirements:

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

These characteristics present a challenge in designing and implementing dependable systems. Since most real-time reactive systems operate in *safety-critical* environments. both functional and timing properties must be analyzed to ensure the satisfaction of safety requirements before deploying the system. Rigorous techniques are necessary to model. design. and analyse the system behavior before committing to an implementation. Although formal notations can adequately deal with complexity in modeling and design and further lead to rigorous validation and verification of the modeled systems, formal notations may be daunting for a developer of the system. When developing such complex systems. visual models provide more clarity and comprehension of modeling elements, subsystem configurations. and interfaces. By providing a visual modeling technique and linking it to formal notations in the background, one achieves the twin goals: easy to use interface for application designers. and a basis for rigorous analysis of the modeled systems, for which the developer does not have to learn the underlying formalism. It is in this context that this thesis is developed.

The foundational work on reactive system modeling is described in [Ach95]. It introduces the *Timed Reactive Object Model* formalism. and the notion of an abstract (generic) reactive model. A complete semantics of the formalism. and several case studies illustrating the expressiveness of the formalism, have appeared in [Ach95] and [AAR95]. TROMLAB [AAM96] is a development environment for real-time reactive systems based on the TROM formalism; Figure 1 is an overall architectural view of TROMLAB.

The following components of the environment are currently operational:

- **Interpreter** - [Tao96] which parses, syntactically checks a specification and constructs an internal representation;
- **Simulator** - [Mut96] which animates a subsystem based on the internal representation. and enables a systematic validation of the specified system:
- **Browser for Reuse** - [Nag99] which is an interface to a library. to help users navigate. query and access various system components for reuse during system development.

The components that are nearing completion are:

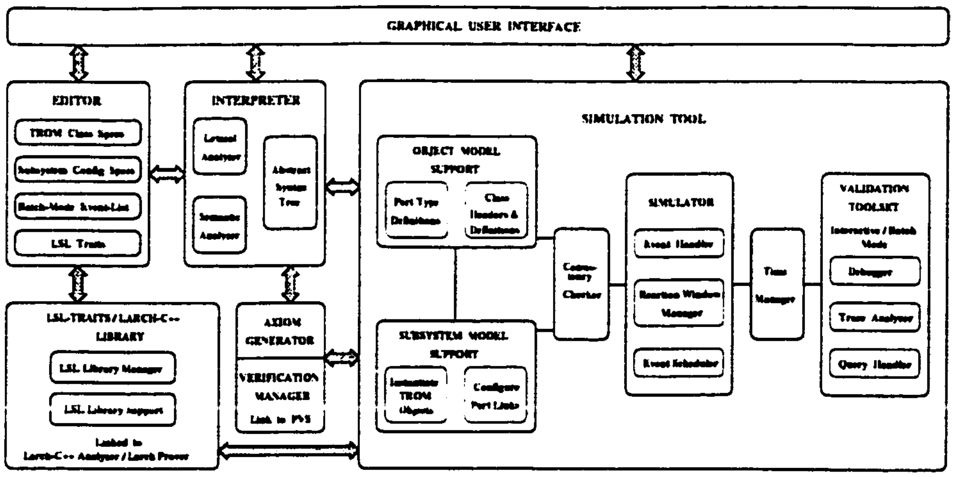


Figure 1: Existing TROMLAB architecture

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

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

Muthiayen [Mut98] in his doctoral dissertation symposium paper has outlined his ongoing research work in integrating object-oriented methodology as practised through the Unified Modeling Language (UML) [Rat97] notations in industries and formal verification approaches based on the formal specification language Prototype Verification System (PVS) [ORS92]. As a first step towards fulfilling this goal, Alagar and Muthiayen [AM98] have proposed minimum extensions to UML notations to model real-time reactive systems as conceived in the generic TROM formalism. This work has extended the UML notation for classes using stereotypes and provided semantics for these new classifier types. By developing class diagrams, state diagrams, and collaboration diagrams to model static and dynamic properties of reactive systems, they have laid the foundation for a smooth integration of visual models to formal notations.

It is in this context that this thesis has made a significant contribution—showing how to realize this method in practice using the Rational Rose 98[1] tool. Starting from the proposal [AM98], and investigating it for Rose implementation, brought about several improvements, which have been incorporated in the model. This thesis presents the features of the revised UML model and its implementation in Rose.

The goal of the present work is to provide a Rose interface for visual modeling of real-time reactive systems, where the elements of the visual model can be automatically translated into the TROM models. This graphical interface is an alternate user interface for the TROMLAB environment. The thesis presents the design and implementation of an easy-to-use interface, a translator to map the visual models into TROM models, and a graphical front-end to a mechanical verifier based on the PVS verification system.

The main research contributions of this thesis are:

- An improved minimal extension to UML notation to capture a generic reactive system model;
- A mapping from the extended UML notation to a formal notation;
- The development of a tool for automatic translation from UML notation to the formal notation;
- An improved UML and formal model for the Train-Gate-Controller problem;
- A UML and formal model for the Mobile Originating Short Message Services industrial application.

Alagar and Muthiayen [AM98] discuss the differences between our UML model for generic reactive systems and other technologies such as DisCo [JKSSS90] and ROOM [SGW94] and [RS98]. Selic and Rumbaugh [RS98] discuss the embedding of ROOM into UML, thus creating UML for Real-Time [Obj98]. UML for Real-Time is an extension to the UML 1.1 visual modeling language that has been specifically fine-tuned

[1]A licence of Rose 98 Enterprise Edition is provided by Ericsson Research Canada as a contribution to this work.

for the development of complex, event-driven, real-time systems. Collaboration diagrams specify the structure of software components by using the primary constructs of capsules, ports and connectors. Finite state machines capture the functionality of simple capsules. The ObjecTime Developer tool from ObjecTime Limited incorporates these UML for Real-Time concepts [Obj97]. Our UML model for generic reactive systems has the TROM technique as a basis, with a well-defined semantics, where requirements validation can be performed at a very early phase, prior to any implementation decisions. Our UML model is an extension of UML 1.1, as supported by Rose 98 from Rational Software Corporation.

The thesis is organized as follows. Chapter 2 presents the notion of a generic reactive system and the formalism. Chapter 3 briefly introduces Rose, the tool used for creating and maintaining UML models for generic reactive systems. Chapter 4 presents a UML modeling technique specific for generic reactive systems and its implementation in Rose. Chapter 5 introduces an automatic translation tool from a Rose model into a formal notation. Chapter 6 presents the Train-Gate-Controller as a case study. A second case study, from the mobile telephony industry, is presented in chapter 7. Chapter 8 suggests future enhancements to the formalism and the translation tool: it describes how the translator may be integrated into TROMLAB, and bridges the gap between the UML model and its future use for formal verification. The thesis ends with a conclusion in chapter 9.

Chapter 2

Formal Model for Generic Reactive Systems

2.1 Introduction

This chapter is a brief survey of the basics of generic reactive systems, introducing the concepts and terminology used in the rest of this thesis.

An object-oriented modeling technique for real-time reactive systems was introduced in [Ach95]. It introduces the *Timed Reactive Object Model* formalism, and the notion of an abstract (generic) reactive model. A complete semantics of the formalism, and several case studies illustrating the expressiveness of the formalism have appeared in [Ach95, AAR95].

2.2 The Informal Model

A generic reactive class (GRC) [AM98] is a visual representation of the *Timed Reactive Object Model* formalism [Ach95]. It is a hierarchical finite state machine augmented with ports, attributes, logical assertions on the attributes and time constraints. Such an object is assumed to have a single thread of control. A GRC communicates with its environment by synchronous message passing, which occurs at a port.

Informally, a reactive object consists of the following elements:

- **A set of events** partitioned into internal, input and output events. *Input and*

Output events occur at a *port* and represent message passing. The names of these events are suffixed by ? and !, respectively. *Internal events* are assumed to occur at the *null* port.

- **A set of states.** A *state* can be simple or complex, and a *complex state* may be decomposed into *substates.*
- **A set of typed attributes.** An *attribute* can be of one of the following two types: an abstract data type specifying a data model or a port reference type.
- **An attribute function.** The *attribute function* defines the association of *attributes* to *states.* For a computation associated with a transition entering a state, only the attributes associated with that state are modifiable and all other attributes will be read-only in that computation.
- **A set of transition specifications.** Each specification describes the computational step associated with the occurrence of an *event.* A *transition specification* has three logical assertions: an *enabling and a post-condition* as in Hoare logic, and a *port-condition* specifying the port at which the transition can occur. The assertions may involve *attributes* and the keyword *pid* for port identifier.
- **A set of timing constraints.** A *timing constraint* can be associated with a transition to describe the time-constrained response to a stimulus. A timing constraint captures the *event* corresponding to the response, *lower and upper bounds* for the time interval during which the event should occur, as well as a list of disabling states. An enabled reaction is disabled when the objects enters any of the *disabling states.*

Figure 2 illustrates the elements of a reactive object.

2.3 The Formal Model

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

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

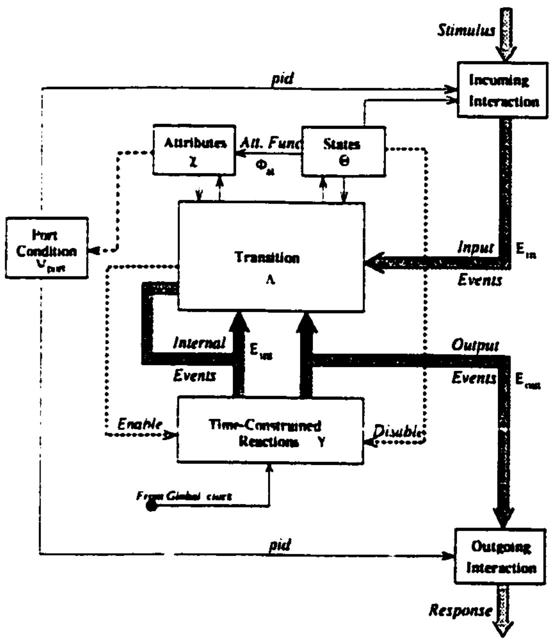


Figure 2: Anatomy of a reactive object

- $\mathcal{P}$ is a finite set of port-types with a finite set of ports associated with each port-type. A distinguished port-type is the null-type $P_\circ$ whose only port is the null port $\circ$.

- $\mathcal{E}$ is a finite set of events and includes the silent-event tick. The set $\mathcal{E} - \{\text{tick}\}$ is partitioned into three disjoint subsets: $\mathcal{E}_{in}$ is the set of input events, $\mathcal{E}_{out}$ is the set of output events. and $\mathcal{E}_{int}$ is the set of internal events. Each $e \in (\mathcal{E}_{in} \cup \mathcal{E}_{out})$, is associated with a unique port-type $P \in \mathcal{P} - \{P_\circ\}$.

- Θ is a finite set of states. $\theta_0 \in \Theta$, is the *initial* state.

- $\mathcal{X}$ is a finite set of typed attributes. The attributes can be of one of the following two types: i) an abstract data type specification of a data model: ii) a port reference type.

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

- Φ is a function-vector (Φ_s, Φ_{at}) where,

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

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

- Λ is a finite set of *transition specifications* including λ_{init}. A transition specification $\lambda \in \Lambda - \{\lambda_{init}\}$, is a three-tuple : $< \langle \theta, \theta' \rangle;\ e(\varphi_{port}); \varphi_{en} \Longrightarrow \varphi_{post} >$; where:

  - $\theta, \theta' \in \Theta$ are the source and destination states of the transition;
  - event $e \in \mathcal{E}$ labels the transition; φ_{port} is an assertion on the attributes in $\mathcal{X}$ and a reserved variable pid, which signifies the identifier of the port at which an interaction associated with the transition can occur. If $e \in \mathcal{E}_{int} \cup \{\mathsf{tick}\}$, then the assertion φ_{port} is absent and e is assumed to occur at the null-port o.
  - φ_{en} is the enabling condition and φ_{post} is the postcondition of the transition. φ_{en} is an assertion on the attributes in $\mathcal{X}$ specifying the condition under which the transition is enabled. φ_{post} is an assertion on the attributes in $\mathcal{X}$, primed attributes in $\Phi_{at}(\theta')$ and the variable pid and it implicitly specifies the data computation associated with the transition.

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

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

The initial-transition λ_{init} is such that $\lambda_{init} : \langle \theta_0 \rangle;\ Create();\ \varphi_{init}$ where φ_{init} is an assertion on active-attributes of θ_0.

- Υ is a finite set of *time-constraints*. A timing constraint $v_i \in \Upsilon$ is a tuple $(\lambda_i, e_i', [l, u], \Theta_i)$ where,

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

A *Subsystem Configuration Specification* (SCS) is defined to specify a system or a subsystem by composing reactive objects or by composing smaller subsystems.

The grammar of the formal specification. based on the above formalism is given in chapter 5.

Figure 3 shows the template for a class specification. Figure 4 shows the template for a subsystem configuration specification.

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

Figure 3: Template for System Configuration Specification.

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

Figure 4: Template for System Configuration Specification.

Chapter 3

About Rational Rose

3.1 Introduction

The Unified Modeling Language (UML) is a notation that combines the best features from data modeling concepts, business modeling, object modeling and component modeling. UML became the standard language for visualizing, specifying, constructing and documenting the artifacts of a software system and was approved by the Object Management Group (OMG) in September 1997.

We chose Rational Rose 98, developed by the Rational Software Corporation, to model generic reactive systems because it is the leading tool for visual modeling using the UML notation. In this chapter we discuss some of its features, mainly those that we use in modeling generic reactive systems and we provide basic information for using Rose.

3.2 UML Features in Rose

Visual modeling is a way to model software systems using a standard graphical notation. A graphical model helps the designer capture the essential part of a system. from different perspectives. With *use case analysis*, Rose allows capturing the *business process* from the end user's perspective, using real-life entities as actors. A visual model bridges the communication gap between the business-oriented end-users and the software developers. Rose facilitates complexity management with the introduction of *packages*, allowing a large system to be broken down into subsystems. Static structures of a system are captured in *class diagrams.* System behaviour is captured in *statechart diagrams.* Interaction between objects may be shown in *collaboration and sequence diagrams.*

Rose has the capacity to *generate code*, that is component and class skeletons and to reverse-engineer an application. These features are not addressed in this thesis.

Rose supports the UML notation and may be *integrated* with other tools from the Rational Suite. for version control. document generation. requirements management. It supports team development by allowing certain parts of the model to be developed separately.

Rose has a *graphical user interface.* which shows different views of the model. Given its ease of use and its versatility. Rose allows the developers to be application specialists rather than methodology specialists.

UML is *not a formal notation.* and therefore Rose does not have a strong formalism. We will use a translating tool to extract information from a Rose model and convert it into specifications in a formal notation with well-defined semantics.

In this thesis we propose to use Rose primarily to model the design of real-time reactive systems. In the following chapters we will refer to Rose's class diagrams. statechart diagrams. collaboration diagrams and sequence diagrams.

3.3 Using Rose

The next few sections contain extracts from [Rat98b]. sufficient information for getting started and creating a Rose model for a real-time reactive system. Only information relevant for the rest of this thesis is provided.

User Interface Rose provides a modern. intuitive graphical user interface on Windows and Unix platforms. The application window. shown in Figure 5. contains a menu control box. a menu bar. a toolbar. a minimize button. a maximize button. the browser. and the documentation window.

The **toolbar** contains different icons. depending on the diagram window open. It contains icons for creating new models, opening existing models. saving the current

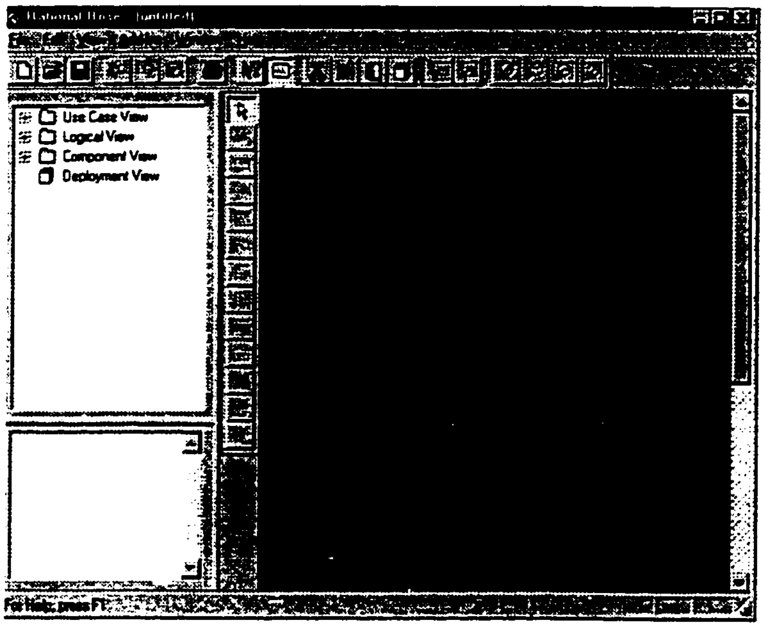


Figure 5: Rose's application window

model or log, editing commands, printing commands, help, displaying dialog boxes for selecting different types of diagrams.

The **browser** is an easy-to-use alternative to menus and toolbars for visualizing, navigating and manipulating items within the model. The browser provides:

- a hierarchical view of all the items in a model.
- drag and drop capabilities that change a model's characteristics.
- automatic updating of model items to reflect changes in the browser.

To hide or display the browser window, click on Browser in the View Menu.

The **documentation window** provides the user with a way to view and modify the documentation of the selected item, which can be either a diagram or a model element.

A **stereotype** is a modeling element subclassification that has been given a more specific meaning. Stereotypes can be applied to packages, classes, attributes, associations. A stereotype can be depicted by either a name or by an icon.

Diagram toolbox

When a modifiable diagram window is active, a toolbox with tools appropriate for the current diagram is displayed. For example, for a sequence diagram, the following tools are available: selector, note, object, message to self, text, note anchor, note, lock.

Creating diagrams

1. On the Browse menu, click xxx Diagram, where xxx is the diagram type
2. In the resulting dialog box, select a view from the list on the left
3. Select $< new >$ from the list on the right and click OK
4. Type the diagram title and click OK

Displaying diagrams

1. On the Browse menu, click xxx Diagram, where xxx is the diagram type
2. In the resulting dialog box, select a package from the list on the left
3. Select a diagram from the list on the right and click OK

Creating an element on a Diagram

1. Click on the appropriate creation tool
2. Click on a location in the diagram

Creating an element in the Browser

1. Click on the appropriate package
2. From the shortcut menu select New, then point to element type (class, package, etc.)

Manipulating icons on diagrams Rose provides similar features to most major drawing tools, including selecting, deselecting, moving and resizing icons.

Specifications A specification enables the user to display and modify the properties and relationships of a model element. Some of the information displayed in a specification can also be displayed inside icons representing the element in diagrams. These fields are standard interface elements such as text boxes. list boxes. option buttons and check boxes. Specification dialogs for some model elements will be shown throughout this text. as appropriate.

Class diagram Class diagrams provide a logical view of the model. displaying icons representing logical packages, classes and relationships contained in the model. There are three ways to create a class diagram:

- On the Browse menu. click Class Diagram
- On the toolbar. double-click the class diagram icon
- On the Browser. double-click on the class diagram icon

Collaboration and sequence diagram In Rose a collaboration diagram is an interaction diagram that shows a sequence of interactions between objects. For the purpose of real-time reactive systems. we propose to use only the static aspect that can be captured in a collaboration diagram. which is the definition of objects and the links between them. as explained in Section 4.13.

In Rose. a sequence diagram is a type of interaction diagram that shows a sequence of interactions between objects.

Rose provides a facility to convert between these two types of interaction diagrams. For the purpose of real-time reactive systems. we do not propose to use this facility.

Statechart diagrams A statechart diagram shows the states of a given class. the events that cause a transition from one state to another. and the actions that result from a state change. Each statechart diagram is associated with one class. A statechart diagram shows exactly one start state, one or several states and the state transitions between them.

There are several ways to create a statechart diagram:

- On the Browse menu. click statechart diagram
- On the toolbar. double-click the statechart diagram icon
- In the browser. double-click the statechart diagram icon

Chapter 4

Generic Reactive Model in Rational Rose

4.1 Introduction

In this chapter we explain how we model a generic reactive system in Rational Rose. Each component of a generic reactive system. as defined in Chapter 2. is treated in a separate section. We use class diagrams and statechart diagrams to capture properties of generic reactive classes. collaboration diagrams to capture system configurations and sequence diagrams to model specific scenarios.

4.2 Generic Reactive Class

A generic reactive class [AM98] is the basic abstract structure of a reactive system. UML provides an extension mechanism using stereotypes. A stereotyped element is represented in UML as the symbol of the base element, with a keyword string in matching guillemets. placed above the name of the element. We introduce generic reactive classes in Rose as classes with stereotype ≪GRC≫. Graphically, a generic reactive class. referred to as a GRC. is represented as shown in Figure 6.

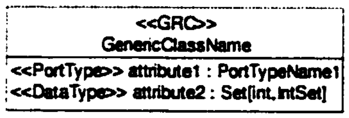
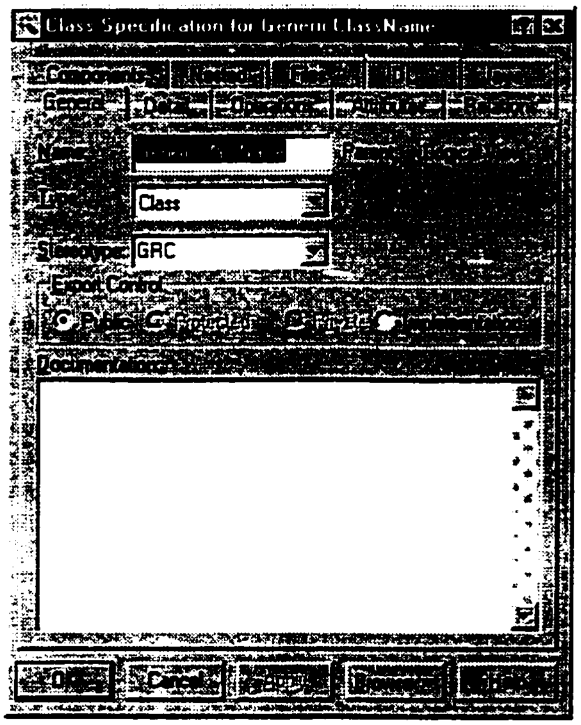

<<GRC>>
GenericClassName
<<PortType>> attribute1 : PortTypeName1
<<DataType>> attribute2 : Set[int, intSet]

Figure 6: Class graphical notation

In order to specify a GRC, we use the facilities provided by Rose for classes: in the browser or directly in a class diagram as introduced in Chapter 3.

The Class Specification dialog, as shown in Figure 7 is used to display and modify GRC properties.


Class Specification for GenericClassName

Type: Class

Stereotype: GRC

Figure 7: Class Specification Dialog in Rose

The class icon has three compartments in Rose: the name compartment, the attribute compartment and the operations compartment. The name compartment is mandatory, but the other two are optional. As will be seen in the following sections, only the attribute compartment is needed for specifying a GRC. Therefore, in order to keep information displayed on a class diagram to the minimum, we suggest suppressing the operations compartment. As defined in UML, if the operations compartment is suppressed, no inference can be drawn as to the presence or absence of operations. We also suggest selecting Rose's option not to show the visibility property of attributes.

A main class diagram should be created, including all the GRCs in a subsystem.

Alternatively, if the system is developed incrementally, a class diagram can be created for each GRC and its associated port types.

4.3 Port Type

A port is an abstraction of an access point for a bi-directional communication channel through which a GRC may interact with its environment. The characteristics of a port are defined by its type. A port-type determines the set of events that can occur at a port of this type.

Following the same extension mechanism with stereotypes as for GRC, we introduce port types in Rose as classes with stereotype ≪PortType≫.

A port type is represented graphically as a class, as shown in Figure 8.

The name of the port type class must start with the symbol "@".

In order to specify a port type class, we use the facilities provided by Rose for classes: in the browser or directly in a class diagram. The Class Specification dialog, shown in Figure 9 is used to display and modify port type's properties.

A port type class must have only one attribute, named events, of type Set. This attribute is considered to be constant. The initial value of this attribute denotes the list of input and output events that can occur at a port of this type. The direction of the events, i.e., input or output, is represented by the symbols "?" and "!", respectively, and it is suffixed to the event name. The list of events has to be given within curly braces, which are the standard notation for sets.

We specify the attribute events using Rose's tab Attributes from the Class Specification dialog and the Attribute Specification dialog, shown in Figure 10.

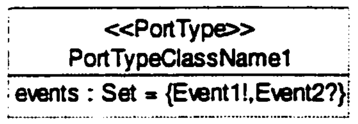
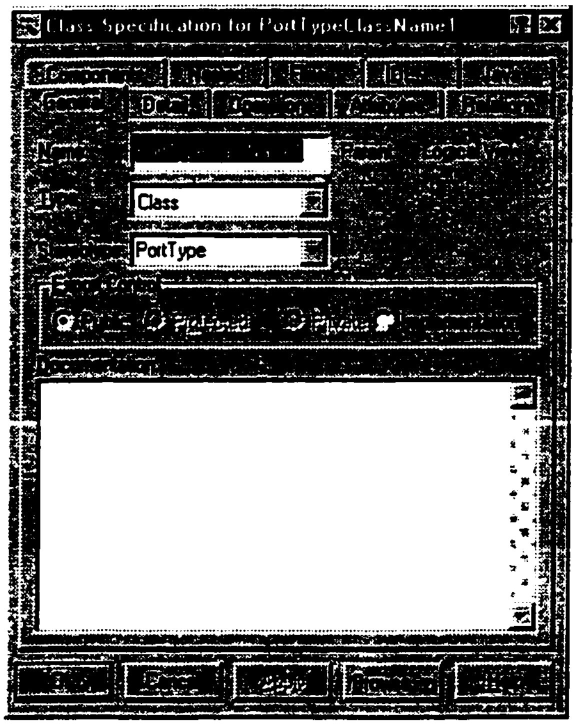


Figure 8: Class graphical notation for a port type with one output event Event1 and one input event Event2.


Class Specification for PortTypeClassName1

Class

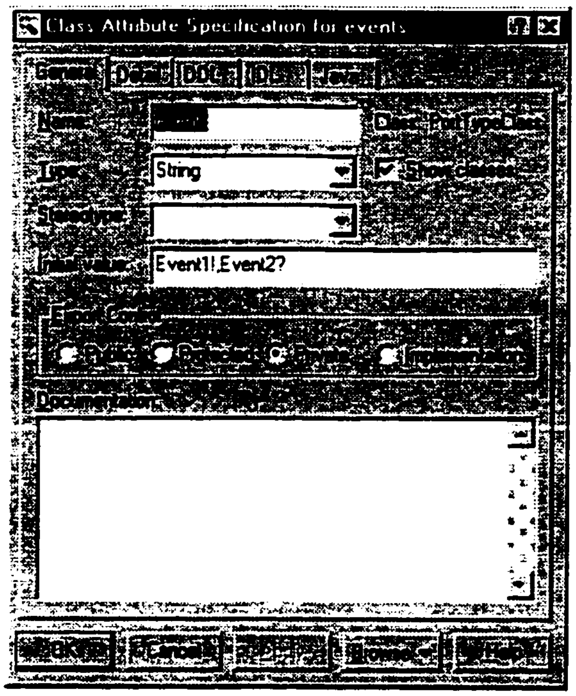
PortType

Figure 9: Class specification dialog for port types

4.4 Relationship between a GRC and its Port Types

A generic class can have several port types. Each port type is used to communicate with another generic class. We represent the relationship between a GRC and its port types as a binary association between the GRC symbol and the port type symbol in a class diagram. The association name is optional. The association end corresponding to the GRC must have the composition aggregation indicator, meaning that the GRC is a composite of port types. The composition aggregation indicator is represented as a filled diamond. Graphically, an aggregation between port type in GRC is represented as shown in Figure 11.

In Rose, in order to obtain the composition aggregation indicator, the "containment by value" property has to be selected in the aggregation specification dialog, detail role tab for the role corresponding to the aggregate class (see Figure 12).



Type: String

Show classes [✓]

Initial value: Event1!,Event2?

Figure 10: Attribute specification for constant attribute events

4.5 Events

In UML, a statechart diagram is a view of the state machine. It shows the sequences of states that an object goes through during its life in response to received stimuli. It also shows the objects responses and actions.

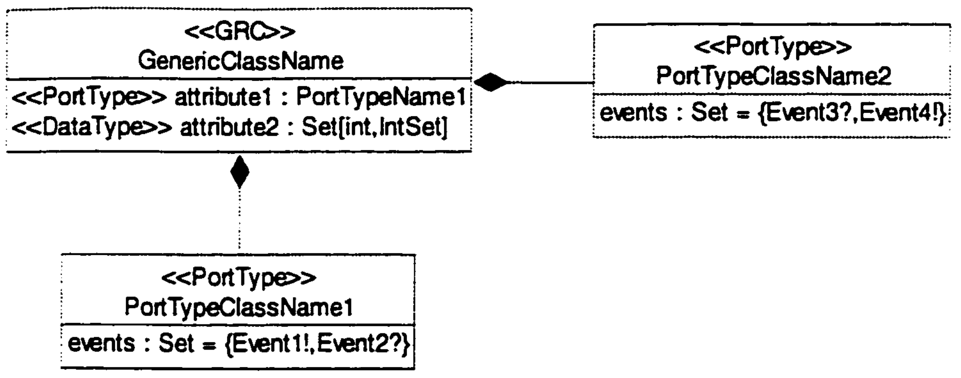
All the events that may occur for a generic reactive class (input, output and internal) are modeled in Rose using the statechart diagram belonging to the GRC.

In Rose, the triggering event is defined as the event that triggers a state transition. Graphically, the event name is represented as the label on the corresponding state transition, as shown in Figure 13.

We define the event using Rose's State Transition Specification dialog, in the field Event (see Figure 14).

Input and output events are also listed in the attribute Events of the corresponding port type classes as described in section 4.3.

If parameterized events were supported in the formal model, the arguments could be entered in Rose in the argument field of the transition specification dialog. See



<<GRC>>
GenericClassName
<<PortType>> attribute1 : PortTypeName1
<<DataType>> attribute2 : Set[int,IntSet]
<<PortType>>
PortTypeClassName2
events : Set = {Event3?,Event4!}
<<PortType>>
PortTypeClassName1
events : Set = {Event1!,Event2?}

Figure 11: Aggregation of two port types to a GRC

Chapter 8 for introduction of parameterized events.

4.6 States

As defined in UML, a state is a condition during the life of an object, during which it satisfies some condition, performs some action or waits for some event. All the states of a generic reactive class are modeled in the statechart diagram in Rose.

A state is represented as a rectangle with rounded corners, as shown in Figure 15. All states of a GRC must have a name. No two states may have the same name, and the same state should be shown only once in the statechart diagram.

We can specify a state in the statechart diagram and by using Rose's State Specification Dialog, Name field (see Figure 16).

UML defines two special states: a start state and an end state. A start state explicitly shows the beginning of the state machine. A start state is connected to the first normal state with an unlabelled transition. Only one start state can exist in each statechart diagram. When applying nested states, one start state should be defined in each context. An end state represents a final or terminal state of a system. An end state is drawn when it is desired to explicitly show the end of the state machine.

For a generic reactive class, we consider the final state of a transition coming from the special start state in Rose as the initial state. We do not use the special end state. Figure 17 shows the Rose start state with a transition to the generic class's initial state.

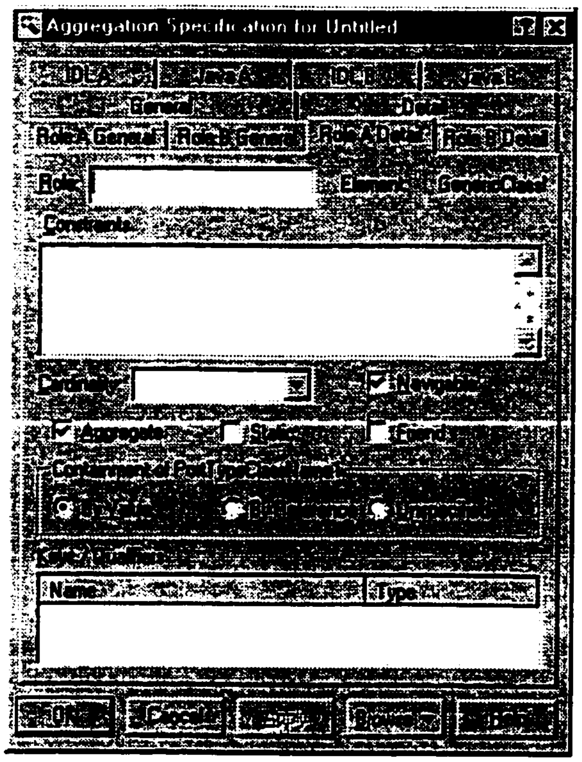


Figure 12: Aggregation Specification dialog, detail role tab

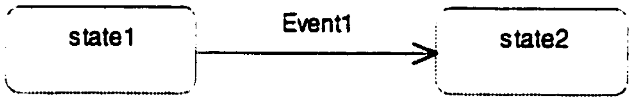


Figure 13: Graphical representation of a event name

4.7 Complex States

In Rose, states may be nested to any depth level. Enclosing states are referred to as superstates, and everything that lies within the bounds of the superstate is referred to as its contents. Nested states are called substates.

We may define complex states for a GRC using Rose's nested states concept. For a generic reactive class, we define an initial state for each level of nesting. Each superstate contains the statechart diagram involving its substates. Figure 18 shows the graphical representation of a complex state.

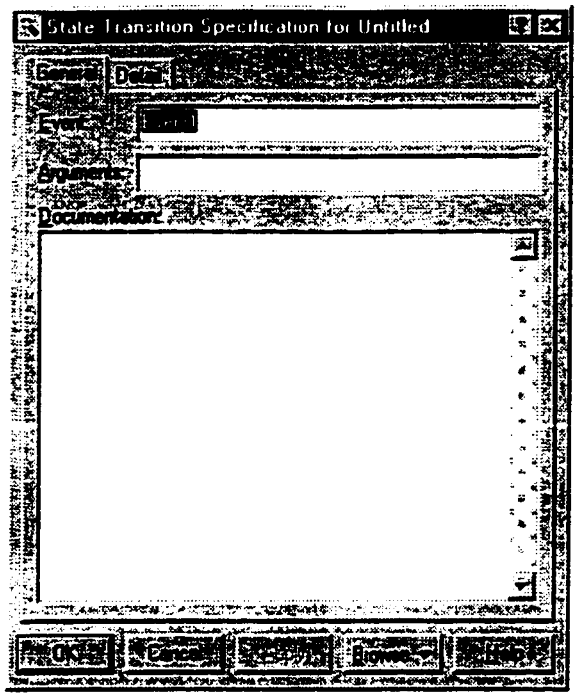


Figure 14: Transition Specification Dialog in Rose

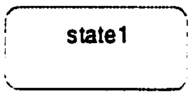


Figure 15: State graphical notation

4.8 Typed Attributes

In UML. the attribute element is used to show attributes of a class. It is semantically equivalent to a composition association.

All typed attributes of a generic reactive class will be represented in Rose as attributes of the GRC. In Rose. an attribute can have a stereotype. an attribute name. a type and an initial value. An initial value is not used in the context of a generic reactive class.

In UML. the type of an attribute is a TypeExpression and it may resolve to a class name. a simple type or a complex type. The details of the type expression are not specified by UML. and may depend on the expression syntax supported by the particular specification [Rat97].

For generic reactive classes. attributes can be of two types: port types and data types.

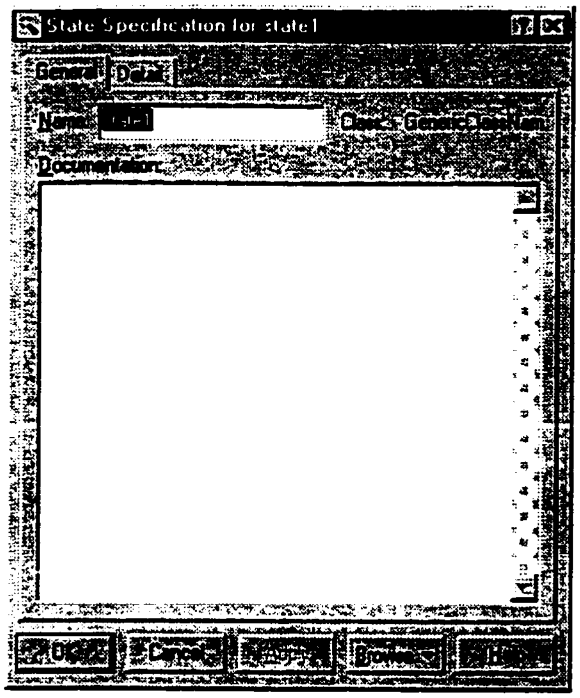


Figure 16: Rose's State specification dialog

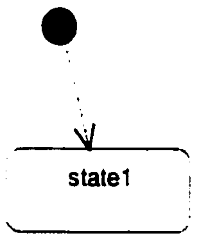


Figure 17: Special start state with the GRC's initial state

Port type attributes are represented in Rose as attributes with the ≪PortType≫ stereotype. The type expression must be the name of a port type class already defined in an aggregate association to the GRC. These attributes may be defined using Rose's attribute specification dialog, shown in Figure 19.

Data types are abstract data types specified in Larch Shared Language traits [GH93]. Data type attributes are represented in Rose as attributes with the ≪DataType≫ stereotype. The type expression may be a simple type, such as Integer or Boolean, or it must be of the form:

<LSL_TraitName>[<parameter1>....<parameterN>,<TypeName>],

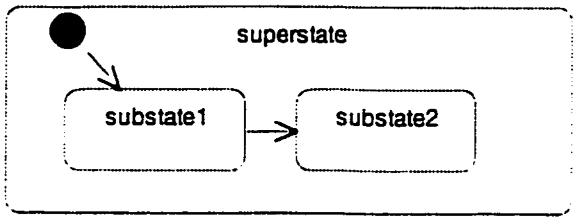
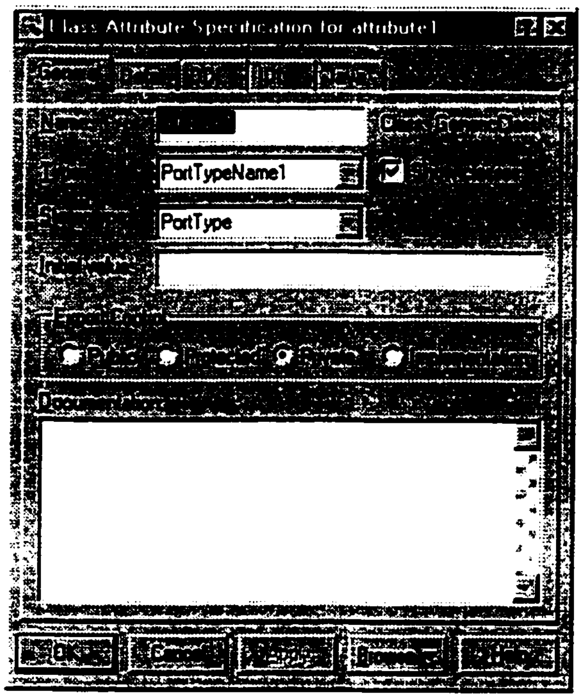


Figure 18: Statechart diagram with nested states



Class Attribute Specification for attribute1

PortTypeName1

PortType

Figure 19: Class Attribute Specification Dialog in Rose for port type attributes

where:

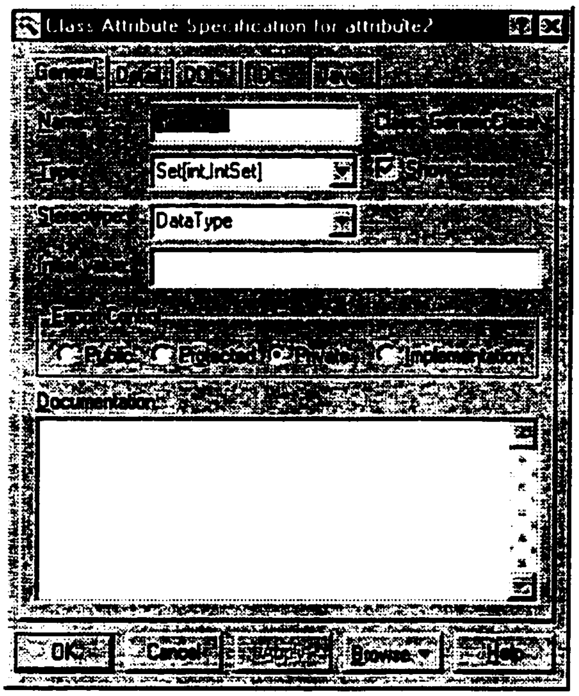
> LSL_TraitName is a string denoting the name of the LSL trait which defines the abstract data type;
>
> Parameter1.....ParameterN, are optional strings, denoting the parameters of the LSL trait;
>
> TypeName, is the name given to the abstract data type, as defined in the LSL trait.

These attributes may be defined using Rose's attribute specification dialog, shown in Figure 20.

When the UML notation is translated to the formal notation, the mapping shown in

Type of attribute	Rose attribute type	Formal attribute type
port type	port type class name	port type class name
data type	LSL trait	TypeName extracted from LSL trait

Table 1: Mapping for attribute types


Class Attribute Specification for attribute2

Set[int,IntSet]

DataType

Figure 20: Class Attribute Specification Dialog in Rose for data type attributes

Table 1 must be done for the type of attributes.

4.9 LSL Traits

LSL Traits introduce the abstract data types used in the data type attributes introduced above. LSL Traits are specified explicitly in the formal notation for a class.

In Rose. the LSL traits are specified as the type expression of a data type attribute. in the following format:

<LSL_TraitName>[<parameter1>,...,<parameterN>,<TypeName>],

where:

LSL_TraitName is a string denoting the name of the LSL trait which defines the abstract data type;

Parameter1.....ParameterN. are optional strings, denoting the parameters of the LSL trait;

TypeName. is the name given to the abstract data type. as defined in the LSL trait.

4.10 Attribute Function

In the formal model, the attribute function associates each state with a set of attributes, possibly empty, denoting the active attributes for that state. By active attributes we mean attributes whose value may change.

In the generic model for UML, we define the attribute function for a state as the set of attributes that are modified in the post-conditions of all transitions incoming in the state. This approach, although different from the user's perspective, does not affect the rigor of the specification or the semantics.

The post-condition of a transition, as described in a later section, is represented in Rose in the Action field of a state transition. Figure 21 shows the graphical representation of a part of the attribute function, where attrib1 is in the attribute function of state2. The Action can be modified in Rose's State Transition Specification dialog, tab Detail, as shown in Figure 22.

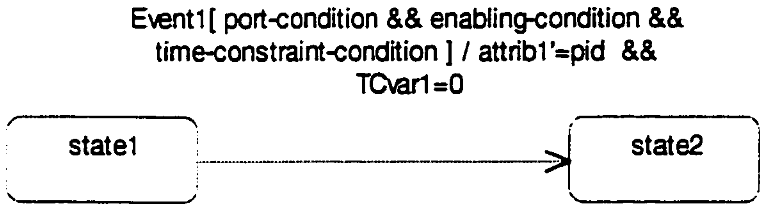
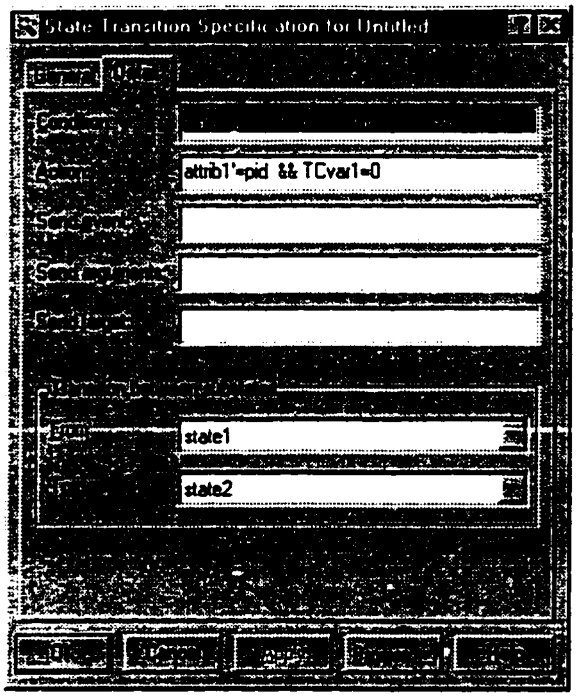

Event1[port-condition && enabling-condition && time-constraint-condition] / attrib1'=pid && TCvar1=0
state1
state2

Figure 21: Graphical representation of attribute function

The post-condition contains operations on attributes. An attribute is said to be modified if the post-condition includes an operation of the form:

attribute_name' = expression,

where the expression yields a result type compatible with the type of the attribute.



State Transition Specification for Untitled

attrib1'=pid && TCvar1=0

state1

state2

Figure 22: Transition Specification, Detail Tab.

4.11 Transition Specifications

In UML, a transition provides a means for tracking an object through a state change; that is, the point at which it is no longer in its source state and has not yet reached its target state.

In UML, a transition is represented graphically as a solid arrow from a state (the source state) to another state (the target state), labelled by a transition string, with the following format:

event–signature [guard–condition] / action-expression ^send–clause

For the purpose of representing transitions for generic reactive classes, the event-signature should contain only the triggering event name. The send-clause is not used. The usage of the guard-condition and action-expression is described below. Figure 23 shows the graphical representation of a transition.

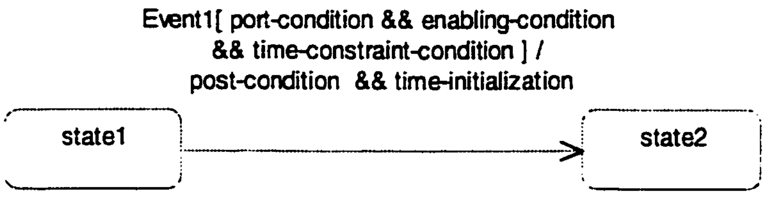

Event1[port-condition && enabling-condition
&& time-constraint-condition] /
post-condition && time-initialization
state1
state2

Figure 23: Transition graphical notation

Transition specifications in the generic model have the following components: initial and final state, triggering event, port-condition, enabling-condition and post-condition.

The port condition is a logical assertion on the attributes of the generic class and the reserved variable pid. If the port condition is true then pid can be bound to any port belonging to the port type of the event associated with the transition. The enabling condition is a logical assertion on the attributes of the generic class and signifies the necessary condition for the transition to take place. The post condition is an assertion on the attributes of the generic class, the primed attributes in the attribute function and the reserved variable pid and signifies the data computation associated with the transition.

These components are modeled in Rose using the statechart diagram, as described in Table 2.

In the generic model	In Rose
Initial State	Source state
Final State	Target state
Triggering event	Triggering event name
Port-condition	First part of the triggering event's guard condition
Enabling-condition	Second part of the triggering event's guard condition
Post-condition	First part of the transition's action

Table 2: Mapping of components of transition specification in formal model and Rose.

The guard condition of a transition has several components: port-condition, enabling condition and time-constraint condition. To facilitate extraction of these components, they have to be separated by the symbol && and specified in a consistent order, as follows:

port-condition && enabling-condition && time-constraint condition.

If the port-condition is not applicable. it should be replaced by the logical expression true.

If the enabling condition is not applicable, it should be replaced by the logical expression true.

The time-constraint condition does not apply to transitions that are not constrained events (see Section 4.12).

If there is no guard condition specified, it is assumed that the port-condition is true, the enabling condition is true and there is no time-constraint.

The action of a transition has the following format:

Post-condition && Time-constraint-initialization.

If the action of a transition is empty, the post-condition is assumed to be the logical expression true.

The time-constraint-initialization does not apply to transitions that are not constraining events. as described in Section 4.12.

All the fields of a transition specification may be entered through Rose's state transition specification dialog. detail tab. as shown in Figure 24.

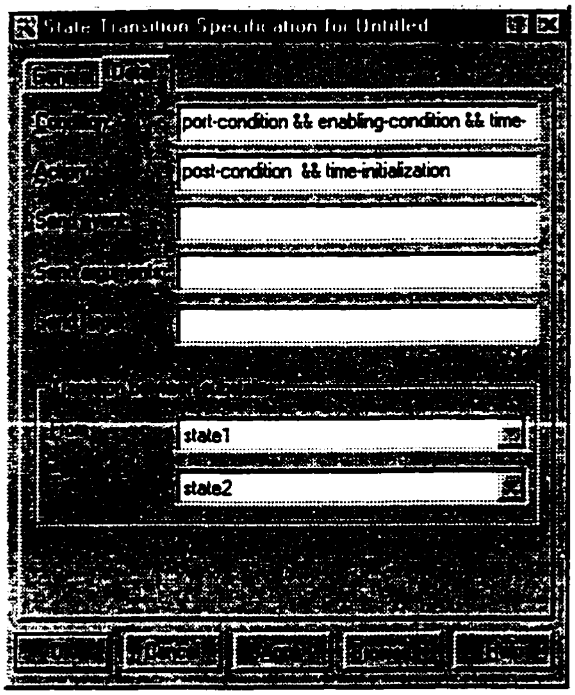
4.12 Time Constraints

In UML there is no predefined way for specifying time constraints.

A time constraint in the generic model has the following components: constraining event. constrained event. lower and upper bounds and a set of disabling states.

Time constraints will be introduced in Rose in the statechart diagram. as follows.

For each time constraint. a reserved variable of type integer will be defined. This variable should be named TCVarN. where N is a numeral (for example TCvar1. TCvar2, etc.). This variable has to be initialized to 0 on the transition of the constraining


State Transition Specification for Untitled

port-condition && enabling-condition && time-

post-condition && time-initialization

state1

state2

Figure 24: Transition specification dialog, detail tab

event. as the second Action. The Action of the constraining transition will thus have two parts. separated by the symbol &&. as follows:

post-condition && time-constraint initialization.

On the transition corresponding to the constrained event, the guard condition has to include the time-constraint condition. as a third predicate. This predicate has the form:

<time-variable> <logical-operator> <lower-bound> &
<time-variable> <logical-operator> <upper-bound>

The condition involving the upper-bound must be specified. If the condition involving the lower-bound is not specified. the lower bound is assumed to be 0. including 0. Logical operators allowed are $\geq$. $\leq$. $<$, $>$. For inequality we use the logical operators $<$ and $>$. For strict inequality we use the logical operators $\leq$ and $\geq$.

The guard condition of the constrained event will thus be of the form:

<Port-condition> && <Enabling-condition> && <Time-constraint-condition>

Figure 25 shows the graphical representation of transitions for the constraining event Event1 and the constrained event Event2, with the time constraint variable TCvar1.

Figures 26 and 27 show the transition specification dialogs for a constraining event Event1 and a constrained event Event2, respectively.

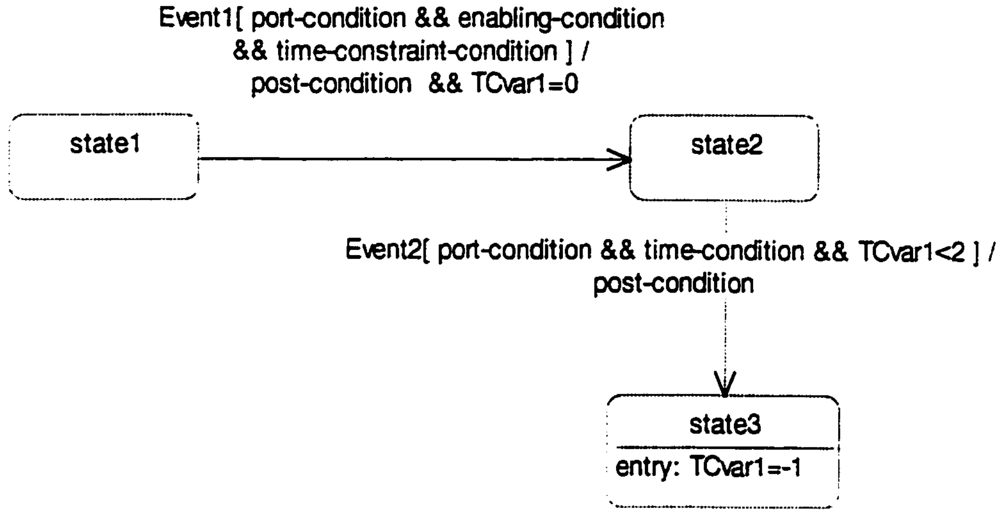

Event1[port-condition && enabling-condition && time-constraint-condition] / post-condition && TCvar1=0
state1
state2
Event2[port-condition && time-condition && TCvar1<2] / post-condition
state3
entry: TCvar1=-1

Figure 25: Graphical representation of constraining event Event1 and constrained event Event2, with time constraint variable TCvar1

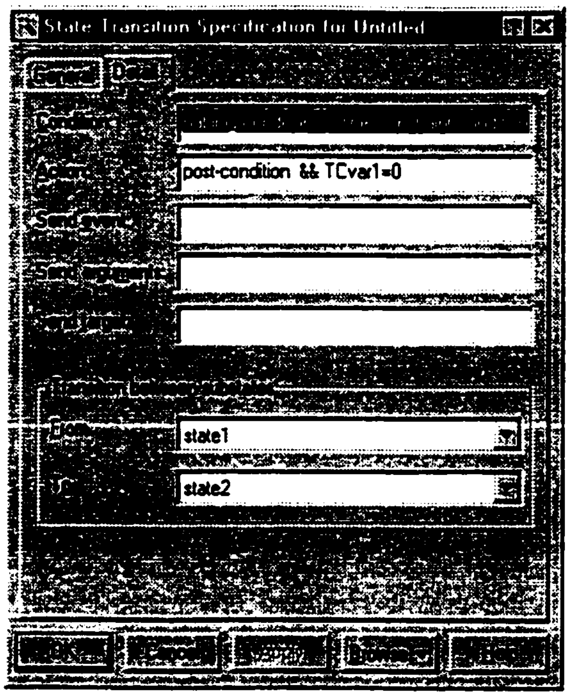
A constrained event may have zero, one or several disabling states. Once the object enters a state that is defined as a disabling state, the constrained event is disabled, that is, it cannot be fired.

In Rose, state actions show what happens upon entering or exiting the state. Actions can be modified using Rose's State Action Specification dialog. Actions on entry are actions that occur when the object enters the state.

If a state is a disabling state for a time constraint, then in the list of Actions on entry of that state, the time variable corresponding to the time constraint will be set to -1. This setting will ensure that the predicate

$$\text{Lower-Bound} < \text{TCvar} \;\&\; \text{TCvar} < \text{Upper-Bound}$$

is false, thus disabling the transition. The set of disabling states for a time constraint


State Transition Specification for Untitled

post-condition && TCvar1=0

state1

state2

Figure 26: Transition specification dialog for a constraining event Event1

will be determined by selecting all the states in which the corresponding time variable is set to -1. The set of disabling states may be empty.

For the purpose of specifying generic reactive systems, no operation other than the time-constraint operation $TCVar = -1$ is allowed among the actions of a state.

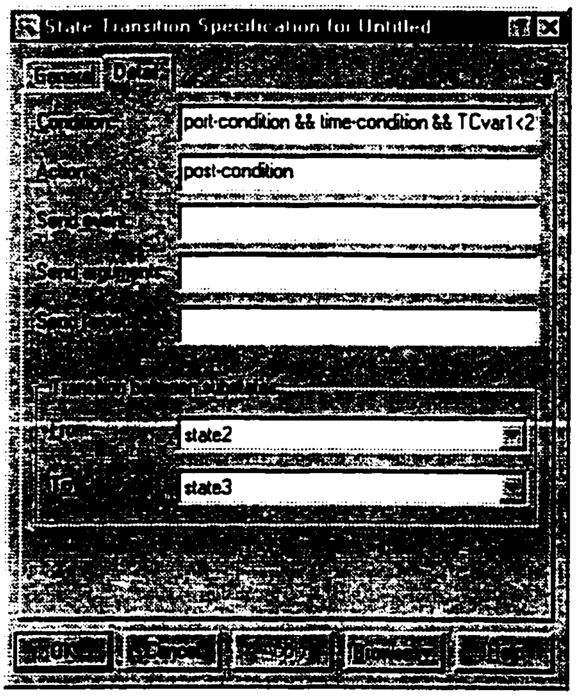
Graphically, a disabling state is shown as a state with the action on entry displayed below the name, as shown in Figure 28.

The action of a state may be modified in the state specification dialog, shown in Figure 29.

Note that the name of the time constraint variable is used to label the time constraint in the formal notation, in order to create a link between the formal and the graphical representation of the model.

4.13 Subsystem Configuration Specifications

A System Configuration Specification is defined to specify a system or a subsystem by composing reactive classes. A subsystem specification consists of three sections:



port-condition && time-condition && TCvar1<2
post-condition

Figure 27: Transition specification dialog for a constrained event Event2

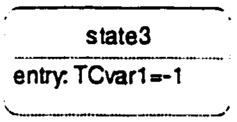


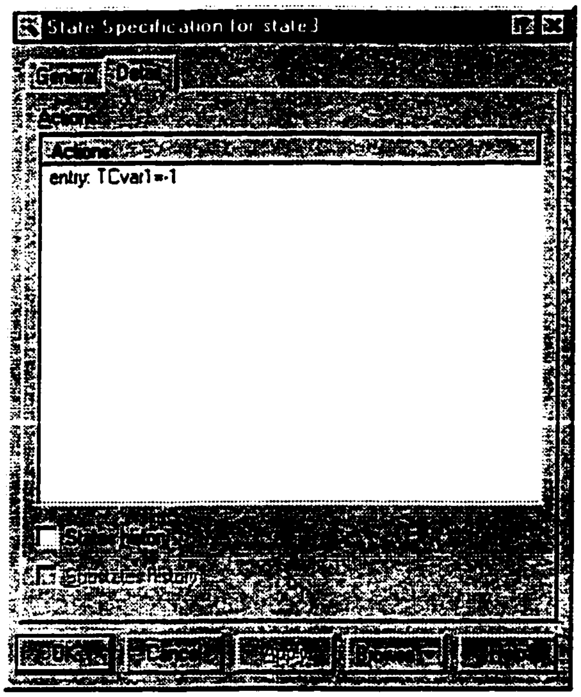
Figure 28: Graphical representation of a disabling state for event Event2.

Include, Instantiate and Configure.

The Include clause is used for importing other subsystems. This section is not used when generic reactive systems are modeled in UML. See section 4.14.

The Instantiate section is used to define generic reactive objects by parametric substitutions to cardinality of ports for each port type and initializing attributes, if any, in the initial state of the object.

The Configure section is used to define a configuration by composing objects specified in the instantiate clause and the subsystem specifications imported through the Include clause. The composition operator $\leftrightarrow$ sets up communication links between compatible ports of interacting objects. Two ports are compatible if the set of input message sequences at one port is a subset of the output message sequences at the



entry: TCvar1=-1

Figure 29: State specification dialog for a disabling state

other port.

In Rose, we model a subsystem as a collaboration diagram. This collaboration diagram contains generic reactive objects instantiated with ports and port links for communication. The purpose of this collaboration diagram is to statically specify the objects, ports and port links, therefore no messages should be shown on the links. Messages on links show interaction between objects.

Interaction between objects shall only be shown in Rose in sequence diagrams. See Section 8.3 for more details on how sequence diagrams are used.

A reactive object is defined in UML as an object of a generic reactive class and is represented graphically as shown in Figure 30. The object may be modified using the object specification dialog, shown in Figure 31.

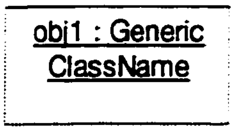
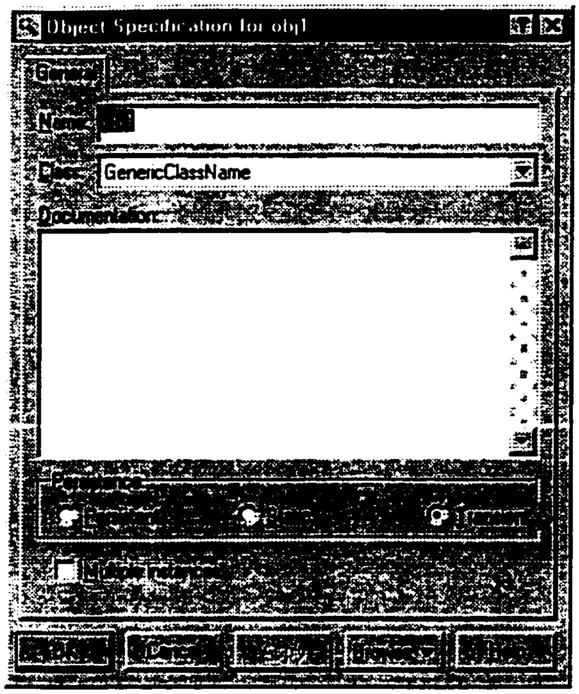


Figure 30: Graphical representation of a generic reactive object



Class: GenericClassName

Figure 31: Object specification dialog

Ports provide the access points for bi-directional communication between the GRC and its environment. A port is defined in UML as an object of the class named with the port type name, and with stereotype ≪PortType≫, and is represented graphically as shown in Figure 32.

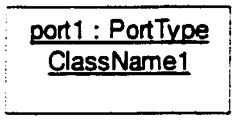


Figure 32: Graphical representation of a GRC object

We define a reserved variable pid as the identifier of the port at which an interaction associated with a transition can occur. This reserved variable can be used in the logical assertions in the port-conditions, enabling-conditions and post-conditions, conform to the formal notation.

An object may have several ports of each port type defined for the class. This relationship is shown as a link between the generic reactive object and the port instances in the collaboration diagram of the subsystem.

Figure 33 shows a generic object with two ports.

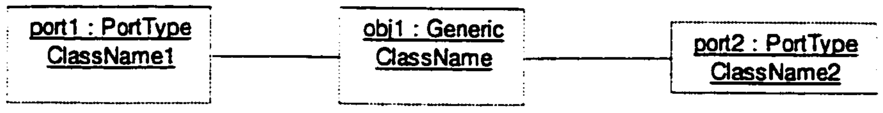


Figure 33: Graphical representation of a generic reactive object with two ports

Reactive objects communicate through ports. A link between two port objects belonging to two reactive objects defines a port link between the two objects.

Figure 34 shows the port link between two reactive objects.

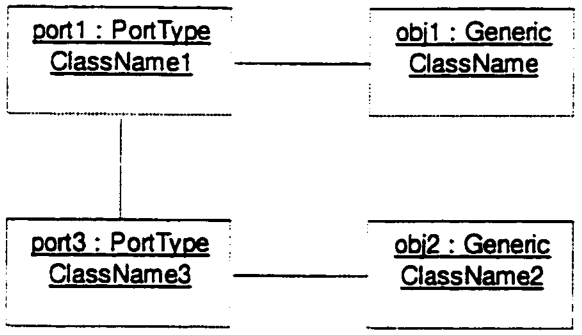


Figure 34: Graphical representation of a port link between two generic reactive objects

4.14 Composing Subsystems

In the previous section we mentioned the Include clause in the subsystem configuration specification. In the formalism. it is used to import objects from other subsystems.

UML does not allow nesting of collaboration diagrams. We propose a way to handle complex systems. based on the way complex systems can be managed in Rose. with packages.

A package is a grouping mechanism that may be used to designate not only logical groupings and physical groupings, but it may also be used to designate use case groupings. processor groups. and distribution units. Each package represents a chunk of the logical architecture of the system [Rat98b].

All the generic reactive classes and their components belonging to one subsystem are contained in a package. If the subsystem should not include objects from other subsystems, the collaboration diagram representing the objects in the subsystem should be placed in the same package as the GRCs. However, if it is desired to import other subsystems, the collaboration diagram should include objects from all the included subsystems. The collaboration diagram should be placed in a package higher in the hierarchy of packages, to indicate that it contains objects from several subsystems.

Chapter 5

Rose-GRC Translator: Visual Models to Formal Specifications

5.1 Introduction

After specifying all the components of a generic reactive class as described in Chapter 4, a translation can be performed automatically from the Rose model to the formal notation. This tool is called Rose-GRC Translator. This chapter presents the requirements. the design and implementation of the translator.

5.2 Requirements

5.2.1 Input: Rose Models

The Rose-GRC Translator shall run in the Rose environment and shall take input from an open Rose model.

Generic reactive class specifications

The Rose-GRC Translator shall take as input one main class diagram or several class diagrams. The user shall be allowed to select the appropriate class diagrams from a list of all the class diagrams in the model, one at a time. A main class diagram shall be selected if all the classes from the subsystem and their port types are shown in that diagram. Separate class diagrams shall be selected if each class and its port

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

Table 3: Grammar for generic reactive class specification

types are shown in a separate class diagram.

Subsystem Configuration Specification

The Rose-GRC Translator shall take as input one or several collaboration diagrams, which include all the objects in one or several subsystems. Given that the model may contain several subsystems, the user shall be allowed to select the appropriate collaboration diagrams, from a list of all collaboration diagrams in the model. The user shall be allowed to enter a name for each subsystem corresponding to the selected diagram. The user shall be allowed to skip translation of a subsystem configuration.

Message Sequencing

The Rose-GRC translator shall take as input one or several sequence diagrams. The user shall be allowed to select the appropriate sequence diagram from a list of all sequence diagrams in the model, one at a time. The user shall be allowed to skip translation of a sequence diagram. The user shall be allowed to enter a time of occurrence for each message in the sequence diagram.

5.2.2 Output: Formal Specifications

Reactive class specifications

The Rose-GRC Translator shall produce as output a text file containing class specifications for all the generic reactive classes (GRC) that are found in the selected class diagram. The class specifications should be according to the grammar described below (Tables 3–11), which is an improved version of the grammar introduced in [Tao96].

A generic class specification (see Table 3) is composed of the class name, events, states, attributes, LSL traits, the attribute functions, transition specifications and time constraints.

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 4: Grammar for generic reactive class title

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 5: Grammar for events

In the grammar, a class (see Table 4) is described by the keyword Class, followed by a string denoting the class name, followed by a list of port types in square brackets . The list of port types is composed of one or several port type names, represented as strings starting with the symbol '@ and separated by a comma.

Events (see Table 5) are introduced by the keyword **Events**, followed by the list of events. The list of events can contain one or several events, separated by comma. Each event can be an internal event, an input event or an output event. Internal events are represented by a string for the event name. Input events are represented by a string as event name, followed by the character ? and the string for the port type at which the event occurs. Output events are represented by a string as event name. followed by the character ! and the string for the port type at which the event occurs.

States (see Table 6) are introduced by the keyword **States**, followed by the state set. The state set is comprised of the initial state, followed by a list of one or several states. separated by comma. A state is represented by a string for the name. If the state is complex, the name is followed by its substates, represented as a state set, within curly braces.

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 6: Grammar for states

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 7: Grammar for attributes

Attributes (see Table 7) are introduced by the keyword **Attributes**, followed by the list of attributes. The list of attributes is comprised of one or several attributes. separated by a semi-colon. Attributes of type port type are represented by a string for the attribute name, followed by colon and by the port type name, which starts with the character @. Attributes of type data type are represented by a string for the attribute name. followed by a colon and by the LSL trait type name.

LSL traits (see Table 8) are introduced by the keyword **Traits**. followed by a list of traits. The list of traits is comprised of one or several traits. A trait is represented as a string for the trait name. followed in square brackets by the argument list and the trait type name. The argument list is comprised of one or several arguments. An argument is either a trait type name or a port type name starting with the character @.

The attribute function (see Table 9) is introduced by the keyword **Attribute-Function**. followed by a list of attribute function applications. The list of attribute function applications has one or several attribute function applications, separated by a semi-colon. Each attribute function application is comprised of the state name as a string, followed by the keyword →, followed by an attribute list, between curly braces. An

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 8: Grammar for LSL traits

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

Table 9: Grammar for attribute functions

attribute list is comprised of zero or several attribute names, separated by a comma.

Transition specifications (see Table 10) are introduced by the keyword Transition-Specifications, followed by the list of transition specifications, separated by semi-colons and new lines. The list of transition specifications is composed of one or several transition specifications, separated by new lines. A transition specification consists of a name, followed by a colon, one or several state pairs, separated by semi-colons, a triggering event, an assertion, the implication operator → and another assertion. A state pair consists of two state names, in brackets, separated by a comma. The triggering event is an event name followed in brackets by an assertion. An assertion is either a simple expression or two simple expressions with a binary operator between them. A binary operator is one of: =, ≠, <, ≤, >, ≥. A simple expression is either a term or two terms with the | logical operator. A term is either a factor, or two factors with the & logical operator. A factor can be the logical operator ! followed by a factor, or the reserved variable pid, or a primed attribute, an attribute, logical expressions *true* or *false*, an LSL term or an assertion in brackets. An LSL term consists of a LSL function name, followed by an argument list in brackets. An argument list is composed of one or several arguments. An argument is either the reserved variable

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> $\rightarrow$ <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	::=	= | $\neq$ | > | $\geq$ | < | $\leq$
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 10: Grammar for transition specifications

pid, or an attribute name or an LSL term. A primed attribute is an attribute (from the attribute function) followed by the character '.

Time constraints (see Table 11) are introduced by the keyword Time-Constraints, followed by one or several constraints, separated by semi-colons and new lines. A constraint has a name followed by colon and the name of the constraining transition specification, the name of the constrained event, the lower and upper bounds, and a list of disabling states. The lower and upper bounds are preceded and followed, respectively, by the open or closed interval indicators. The list of disabling states is comprised of zero, one or several state names, separated by a comma.

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 11: Grammar for time constraints

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

Table 12: Grammar for subsystem configuration

Subsystem Configuration Specification

The Rose-GRC Translator shall produce as output a text file containing the configuration specification of the subsystem modeled in the selected collaboration diagram. One file shall be produced for each collaboration diagram selected. The configuration specification should respect the following grammar, introduced in [Tao96].

A subsystem configuration specification (see Table 12) is introduced by the keyword SCS. followed by its name as a string, a new line and the following sections: **Includes. Instantiates**, **Configure**. all followed by the keyword **end**.

The include section (see Table 13) is introduced by the keyword **Includes**. followed by a list of subsystem names and a new line. The list of subsystem names is composed of one or several subsystem names. separated by a semi-colon.

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

Table 13: Grammar for include section

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

Table 14: Grammar for instantiate section

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

Table 15: Grammar for configure section

The instantiates section (see Table 14) is introduced by the keyword **Instantiate**, followed by an instance list and a new line. An instance list is composed of one or several instances. An instance consists of an object name, followed by two colons, a generic class name and, in square brackets, by a port cardinality list. The port cardinality list is composed of one or several port cardinalities. A port cardinality is represented by a port type name, followed by a colon and a natural number for the cardinality.

The configure section (see Table 15) is introduced by the keyword **Configure**, followed by the object port list. The object port list is composed by one or several object port links, separated by a semi-colon. An object port link is composed of an object name, followed by a period, a port name starting with character @ and its port type, the composition operator $\leftrightarrow$, another object name, followed by a period, and a port name starting with character @ and its port type.

message_list	::=	<message> | <message> NL <message_list>
message	::=	<send_obj> : <messge_name> : <rec_obj> | <send_obj> : <message_name> : <rec_obj> : <time>
Time	::=	String
send_obj	::=	<obj_name>
rec_obj	::=	<obj_name>
message_name	::==	String
obj_name	::=	String

Table 16: Grammar for message sequence

Message Sequencing

The Rose-GRC Translator shall produce as output a text file containing an ordered list of messages together with the sending and receiving objects and the time of occurrence, from a selected sequence diagram. The translator should produce a file for each sequence diagrams selected by the user. The message sequencing output should be formatted according to the following grammar.

A message list (see Table 16) is composed of one or several messages, separated by newlines. A message consists of a sending object, followed by a colon, the message name, colon, and the receiving object. If time of occurrence is specified, then a colon and the time follow the receiving object. The sending object and the receiving objects are object names, represented as strings. The message name is also a string. The time is represented as a string for convenience.

5.2.3 Use Case Analysis

A use case diagram was created as a first step in the analysis of requirements for the Rose-GRC Translator, and is illustrated in Figure 35.

This use case diagram contains three actors: Developer, Rose and TROMLAB. Rose is considered to be an actor because the model is for the Translator, and the Rose GUI and model are outside the translator. The diagram shows two main use cases (Create-RoseModel and TranslateModel), one mandatory use case (TranslateGRCSpec) and two optional use cases (TranslateSCSSpec and TranslateMessageSequence).

The Developer creates a Rose model, and the output is sent to Rose, in the sense that a Rose model file is saved. The Developer translates the model, which at least translates

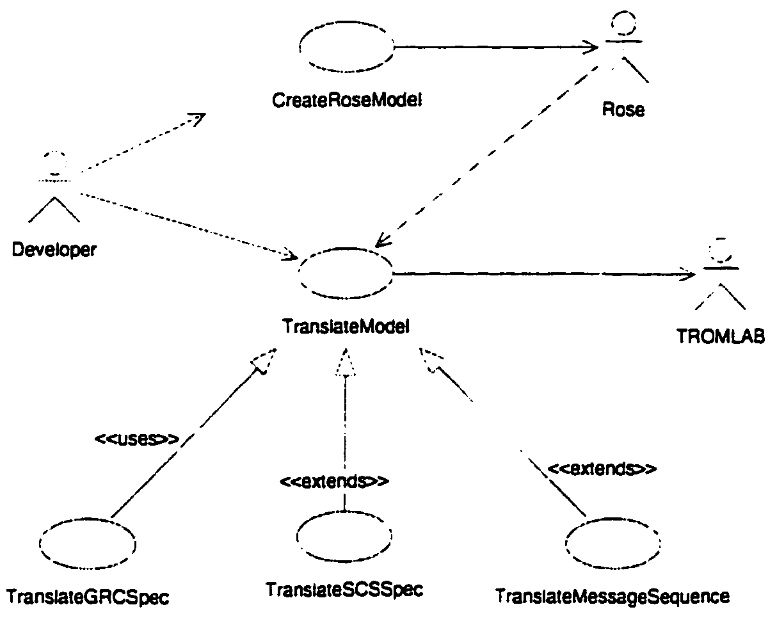


Figure 35: Use Case diagram for the Rose-GRC Translator

a GRC specification. Optionally, subsystem configuration specifications and message sequences may be translated. The output containing formal specifications is sent to the TROMLAB environment.

5.3 Design

5.3.1 Architectural View

The translator is syntax-directed: it takes Rose diagrams and maps the modeling elements onto the corresponding syntactic components as defined by the grammar. The Architecture of the Rose-GRC Translator (Figure 36) is designed in such a way as to emphasize the information flow between its components. The Rose Graphical User Interface is used to create a Rose Model. When a Rose Model is modified, its view is updated on the GUI. The Rose Graphical User Interface triggers the translator's Graphical User Interface. The translator's GUI uses information such as lists of different types of diagrams from the Rose Model. The translator's GUI triggers and controls execution of the translator. The translator sends information, mainly error and control messages, to its GUI. The translator extracts model elements from the Rose Model. The translator creates output files containing formal specifications.

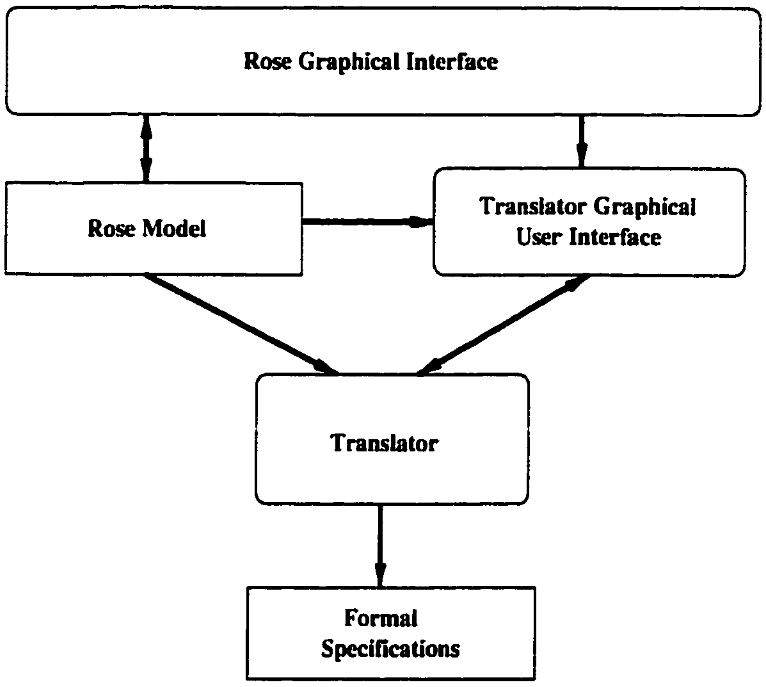


Figure 36: Architecture of the Rose-GRC Translator

5.3.2 Detailed Design

The logic of the translator is built around the components of a generic reactive model, therefore the internal data structures of the translator reflect all these components.

The translator has two parts (Figure 37): the first part is responsible for extracting all the appropriate information from the Rose model, checking its validity and storing it in the translator's internal structures. The second part is responsible for formatting the formal specifications according to the grammar and for creating the output files. This separation may seem redundant at first, but it has a strong motivation: it allows for a thorough error and consistency checking, which ensures that correct formal specifications are produced.

We created a model of the data structure, where each abstract data type is represented as a class. The model is separated into three packages: a model for internal structures for GRC specifications, a model for internal structures for SCS specifications and a model for message sequence entries. Figure 38 shows the class diagram of the translator's internal data structures for GRC specifications. It should be noted that although this diagram is very similar to the Rose model structure, it is just a

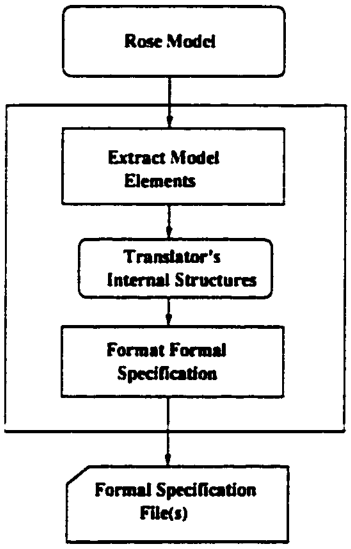

Rose Model
Extract Model Elements
Translator's Internal Structures
Format Formal Specification
Formal Specification File(s)

Figure 37: The two parts and internal structure of the translator

representation of the translator's internal structures that are needed for GRC specifications. The main class in this diagram is the GRC class, which is an aggregation of all the other classes. Figure 39 shows the class diagram of the translator's internal structure for SCS specifications. This model shows two structures: one for all the generic objects and one for all port links of the subsystems. Figure 40 shows the class diagram of the translator's internal structure for message sequences.

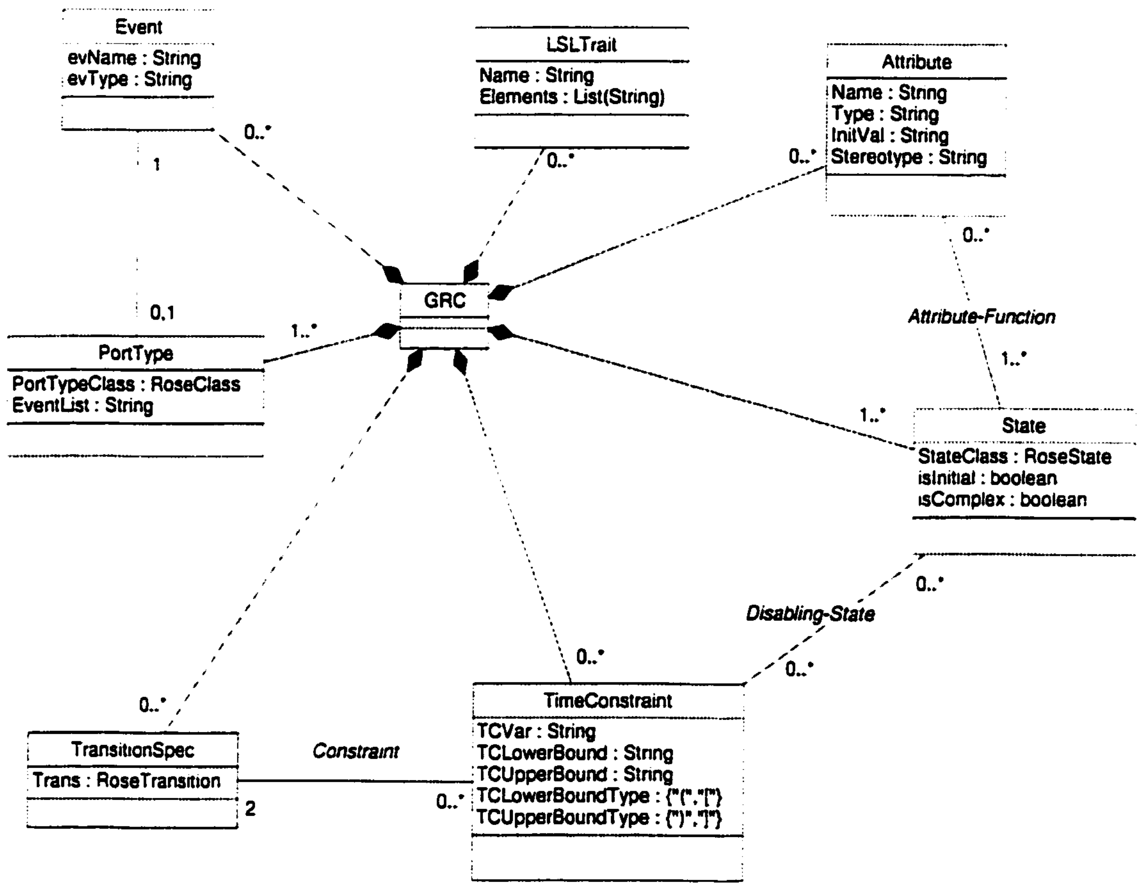


Figure 38: Class diagram for translator's internal structures for GRC specifications

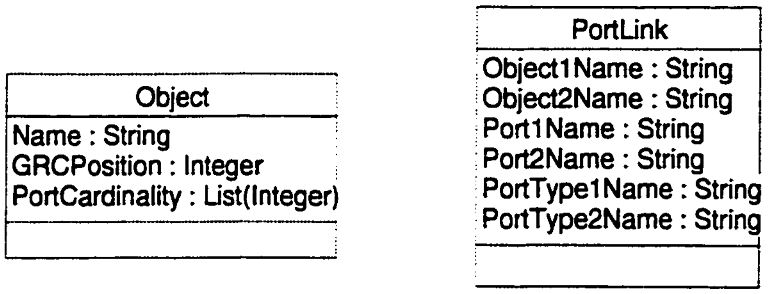


Figure 39: Class diagram for translator's internal structures for SCS specifications

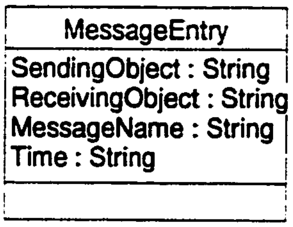


Figure 40: Class diagram for translator's internal structures for message sequences

5.3.3 Algorithms for Extracting Relevant Elements from Rose Model

Main translator logic

```
Get GRC class names
For each GRC class found
    Get port types
    Get events
    Get attributes
    Get states
    Get attribute function
    Get transition specifications
    Get time constraints
End for
Format GRC specifications and output to a file
Get subsystem configuration
Format SCS specification and output to a file
Get message sequence
Format message sequence and output to a file
```

Algorithm for extracting GRC classes from class diagrams

```
For all selected class diagrams returned from GUI
    For all classes from the class diagram
        If stereotype is PortType then skip.
        If stereotype is GRC then store.
        Else Error.
    End for
End for
```

Algorithm for extracting port types

```
Get all the roles of the GRC class
For all roles
    Get the other role of the association
    If other role has stereotype PortType then
        If state machine exists for port Type class then Error.
        If association is not an aggregation then Error.
        If port type class name does not start with @ then Error.
        If port type class has more than one attribute then Error.
        If the attribute of port type class is not named EVENTS then Error.
        If EVENTS attribute of port type class is not of type Set then Error.
        If initial value of EVENTS is not enclosed in {} then Error.
        Store port type class in list of port types for the GRC.
        Store initial value of attribute EVENTS as event set in list of port types for..
        ..the GRC
End for
If no port types found for the GRC class then Error.
```

Algorithm for extracting events and their types

```
If GRC class does not have a state machine then Error.
For all transitions in the state machine
```

```
    skip transition outgoing from Rose's start state
    skip transition incoming into Rose's end state
    If no triggering event defined for transition then Error.
    Store name of event in list of events for the GRC.
    Default event type to internal in list of events for the GRC.
End For
For all port types stored
    If event set is empty then Error.
    For all events in the event set
        If direction ? Or ! Is not specified then Error.
        If this event was not stored in list of events for the GRC then Error.
        Store event direction ? Or ! as event type in list of events for the GRC.
    End for
End for
```

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 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
```

Algorithm for extracting states

If the GRC class does not have a state machine then Error.

```
For all states in the state machine
    If state is Rose's start state or end state then skip.
    Store the state in list of states for the GRC.
    Store indicator for existence of substates in list of states for the GRC.
End for
For all states in the state machine
    If state is Rose's start state then
        Get the transitions of this state from the state machine
        If more than one transition found then Error.
        Get the unique transition.
        Get the target state of the transition
        Store the indication for initial state in list of states for the GRC.
        End for
If no initial state was found then Error.
```

Algorithm for extracting Attribute Function

```
For all transitions in the state machine of the GRC class
    If target state of transition is Rose's end state then skip.
    Get the trigger action of the transition
    If trigger action is empty then skip.
    If trigger action is not of the form Post-condition && Time-initialization ..
        ..then Error.
    Extract post-condition from trigger action
    Extract names of all attributes from left-hand-side of assignments in the ..
        ..post-condition.
    Store names of attributes extracted in attribute function list for the target state ..
        .. of the transition.
End for
```

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.
End for
```

Algorithm for extracting time constraints

```
For all transitions in the state machine of the GRC class
    If guard condition is empty then skip.
    If guard condition is not of the form port-condition && enabling-condition && ..
        ..time-condition Then Error.
    Extract time-conditions from guard condition.
    For all time-conditions extracted (are separated by &)
        If time-variable Name is not of the form TCvar# then error.
        Store time-variable and event name of the transition
        Get lower and upper bounds from the inequality.
        Store lower and upper bounds.
        If time-variable is already in the list of time constraints for the GRC Then Error.
        If lower bound > upper bound then Error.
        For all transitions in the state machine of the GRC class
```

```
          If time-variable is in time-initialization from trigger action then
              If initialization found already then Error.
              If time-variable is not initialized to 0 then Error.
              Set indicator for initialization found.
              Store the event of the transition as constraining event.
      End For
      For all states in the state machine of the GRC class
          Extract state action
          If state action has operation $TCvar = -1$ then store state as disabling ..
              ..state.
          If state action has other operations involving $TCvar$ then Error.
      End for
   End for
End for
```

Algorithm for extracting subsystem configuration

```
For all collaboration diagrams selected from GUI
    For all objects in collaboration diagram
        If no class specified for object then Error.
        If the stereotype of the class is GRC and if the class was stored as GRC..
            .. before then
            Store the GRC object.
            For all links of this object
                If the other role of the link is another GRC object then skip.
                If the other role of the link is a port then
                    If the name of the port does not start with @ then Error.
                    Increment and store the cardinality of the port type corresponding ..
                        ..to this port for the GRC object.
            End for
    End for
    For all links in collaboration diagram
        If both roles of the link are ports then
```

```
            Store the two port names and their port types.
            For both roles of the link
                Get the other link in which this is a role.
                Get the other role of this link.
                If the other role of this link is a GRC object then
                    Store the GRC object.
            End for
    End for
End for
```

Algorithm for extracting message sequence

```
For all sequence diagrams selected from GUI
    For all messages in the sequence diagram
        Store the sending object, the message name and the receiving object
    End for
    For all the stored messages
        Get the time of occurrence from the GUI
    End for
End for
```

5.4 Implementation

5.4.1 Implementation Language

Rational Rose 98, through its Extensibility Interface (REI) [Rat98a] provides a way to extend and customize its capabilities. The language is called RoseScript and it allows accessing Rose classes, properties and methods, updating models and generating documentation.

RoseScript is not an object-oriented language, and in order to implement the design, all classes are replaced by data type structures. The algorithms are implemented as subroutines and functions. The implementation is sequential, with calls to properties and methods of some Rose classes.

The data structures implemented are explained and shown in Appendix A.

We used the following Rose classes and collections: Class, Role, Association, Attribute, Transition, Event, StateMachine, State, Action, ClassDiagram, ScenarioDiagram, Link, ObjectInstance, RoseItem, Message, ClassCollection, RoleCollection, AttributeCollection, TransitionCollection, StateCollection, ActionCollection, ItemCollection, MessageCollection.

Appendix B lists all the properties and methods used in the Rose-GRC Translator.

The biggest problem that we faced when using RoseScript was its lack of user-defined dynamic structures, which imposes a limitation in the number of components of a class. We also experienced some memory allocation problems when the number of components of a class was large, however this was solved by adjusting the size of the translator's internal structures.

5.4.2 File Handling

The Rose-GRC Translator runs in the Rose environment, thus it needs a model file (*.mdl) open in Rose's application window.

The Rose-GRC Translator produces several output files, depending on the user's selections. Each of these files is written according to the grammar listed in the output requirements.

Figure 41 shows the file structure of the Rose-GRC Translator and the relationship between the selected components of the input file and the corresponding output files.

All the generic reactive class specifications are listed in one text file, whose name is entered by the user.

The subsystem configuration specification for each subsystem selected is listed in one text file, whose name is entered by the user.

The message sequencing from each sequence diagram is listed in a separate text file,

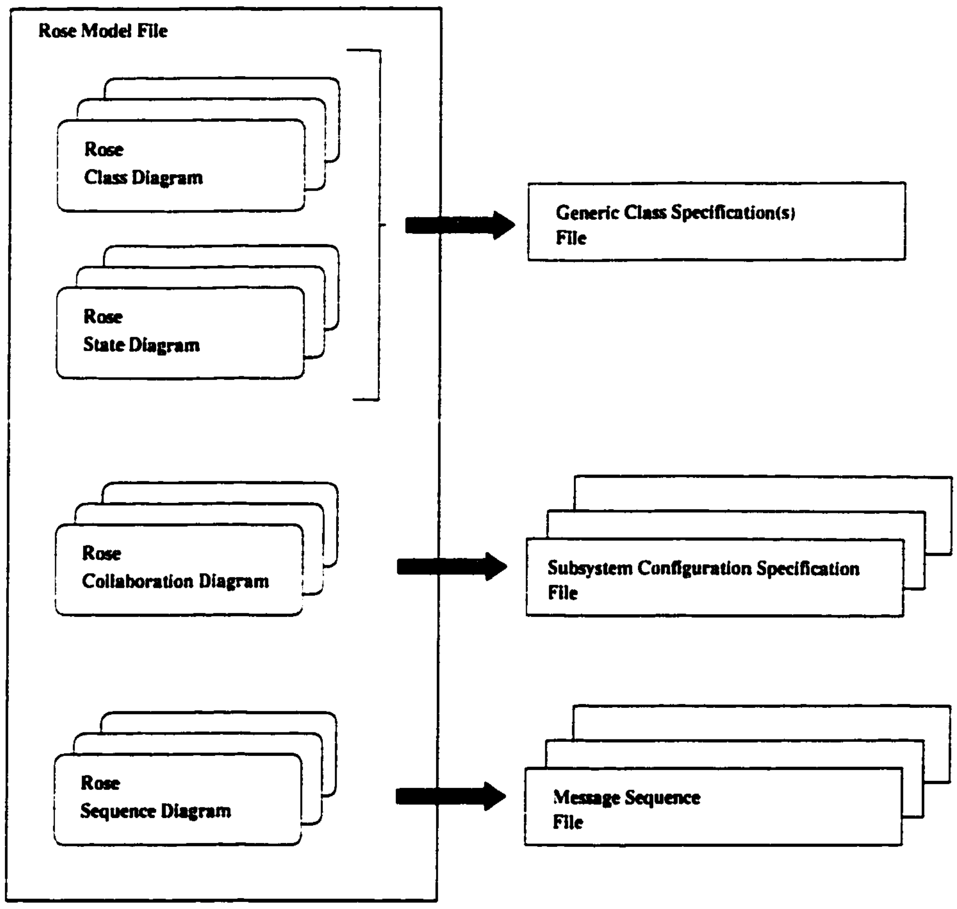


Figure 41: Translator file structure

where each file name is entered by the user.

The user may save the error log window (See section 5.4.3) by using the SavelogAs option from Rose's File menu. This option is available in the File menu only when the log window is open.

5.4.3 User Interface

The Rose-GRC Translator uses a few graphical interface constructs that are provided in the Rose Extensibility Interface: message boxes, dialog boxes, select boxes, a log window and a viewport window. The GUI of the translator is designed in a consistent way with Rose's GUI. It provides a user friendly way to select diagrams and it assists the user in creating a correct model through a comprehensive set of error messages. The Rose-GRC Translator may be executed directly in the Rose environment, being triggered by a Rose menu entry.

Translator as an option in a Rose Menu

We followed these steps as indicated in [Rat98b] to add the translator script to the Rose menu: Follow these steps to add a script to a Rose menu:

1. Open the Rose Menu file (Rose.mnu) in the Rose installation directory.
2. Edit the Path Map so that it includes a virtual script path.
3. Modify the Rose menu file to add the script under the appropriate menu. being careful to follow all of the menu file syntax rules.
4. Save the updated menu file.

The following is the menu entry for the translator. added to the Tools menu:

```
Separator
   Option "Rose-GRC Translator"
      {
        RoseScript $TRANSLATOR-SCRIP-TPATH\Translator.ebs
      }
```

Viewport to show status

The Rose-GRC Translator gives feedback to the user on the status of execution via the viewport window. The viewport window, if not already open. is opened by the Rose-GRC Translator. The viewport window is scrollable. The status messages listed below may be displayed.

Starting the GRC translator...
Reading GRC class names...
Looking for port types for GRC <class name>...
Looking for events for GRC <class name>...
Looking for event types for GRC <class name>...
Looking for attributes for GRC <class name>...
Looking for states for GRC <class name>...
Looking for attribute function for GRC <class name>...
Looking for transition specifications for GRC <class name>...

```
Looking for time constraints for GRC <class name>...
Looking for initialization of <time-constraint variable name>...
Looking for objects and ports in collaboration diagram <collaboration
   diagram name>...
Looking for port links in collaboration diagram <collaboration diagram
   name>...
Extracting message sequence from sequence diagram <sequence diagram
   name >...
```

Rose's log window

Within the Rose environment there is a log window, which may be used to log errors. The Rose-GRC Translator writes an appropriate text in this log window for each error that is detected. When the program terminates, the log window contains all the errors that were detected. Section 5.5 gives a list of error messages.

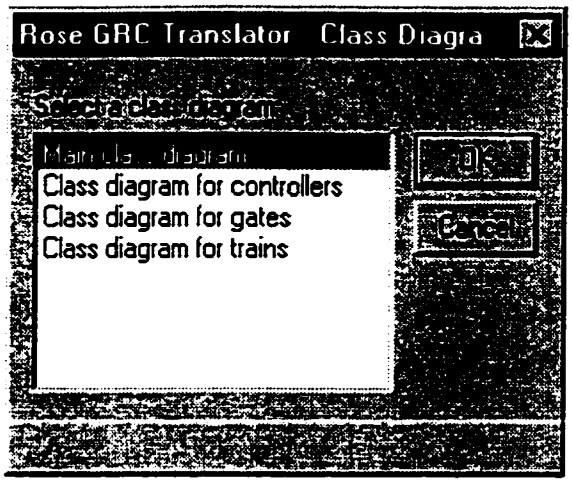
This log window is also used by several commands from Rose to report progress, results and errors. When Rose is started, the log is displayed as an icon at the bottom of the window. Double-click on the icon to open the log and display the contents. When the log window is open, the following choices are available under the File menu: AutoSave Log, Save Log As, Clear Log.

Select boxes for selection of diagrams

The Rose-GRC Translator uses select boxes to allow the user to select diagram names for the following types of diagrams: Class diagrams, collaboration diagrams and sequence diagrams. Figure 42 illustrates the select box for class diagrams.

The select box has title "Rose-GRC translator <diagram type> ". It displays a text asking the user to make a selection. A list of all the diagrams of the given type from the model is displayed. The box also contains an OK button and a Cancel button. The user may highlight any entry in the list by clicking on it with the left mouse button or by moving up and down the list with the arrow buttons. The user may select the highlighted entry by pressing the OK button or double-clicking the left mouse button on it. If the user presses the cancel button, the program is terminated.

Message Box for yes/no questions

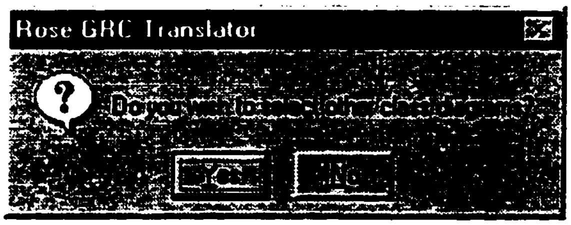

Rose GRC Translator Class Diagra

Select a class diagram:

Main class diagram
Class diagram for controllers
Class diagram for gates
Class diagram for trains

Figure 42: Select box for class diagrams for Train-Gate-Controller model

The Rose-GRC Translator uses message boxes in order to ask the user simple yes/no questions. These message boxes have a text, a Yes button and a No button and a Question Icon. The text is dependent on the logic. One such message box is shown in Figure 43. After the user presses either the Yes or the No button, execution continues depending on the logic.


Rose GRC Translator

Do you want to select other class diagrams?

Figure 43: Message Box asking to continue execution of the program

Message Box for error handling

For every error that is detected by the Rose-GRC Translator, a message box is displayed. One such message box is shown in Figure 44.

The message box has the title "Rose-GRC Translator Error", has a text, an OK button and a Cancel button. The text is dependent on the error that has occurred. Section 5.5 gives a list of error messages. If the user presses the OK button, execution continues according to the logic of the translation algorithms, depending on the type of the error. Please see the Error handling section. If the user presses the Cancel button, the program is terminated immediately.

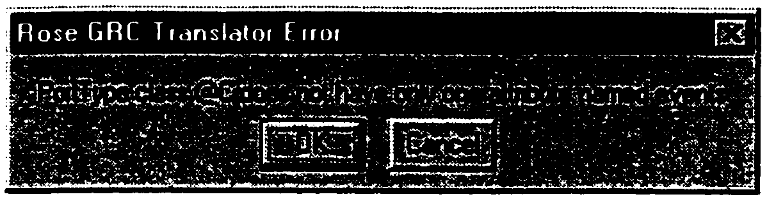
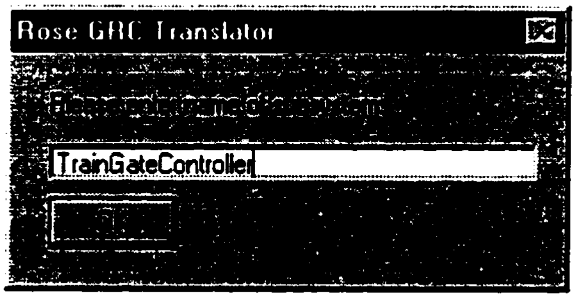


Figure 44: Message Box for reporting an error

Dialog Box for Subsystem Name

A dialog box, shown in Figure 45, is used for allowing the user to enter the name of the subsystem for which to create a subsystem configuration specification. The dialog box has the title "Rose-GRC Translator". There is a text asking the user to enter the subsystem name, followed by a text input field. The user has to click the left mouse button on this field before entering the subsystem name. The dialog box contains an OK button. When the user presses the OK button, execution of the program continues.



TrainGateController

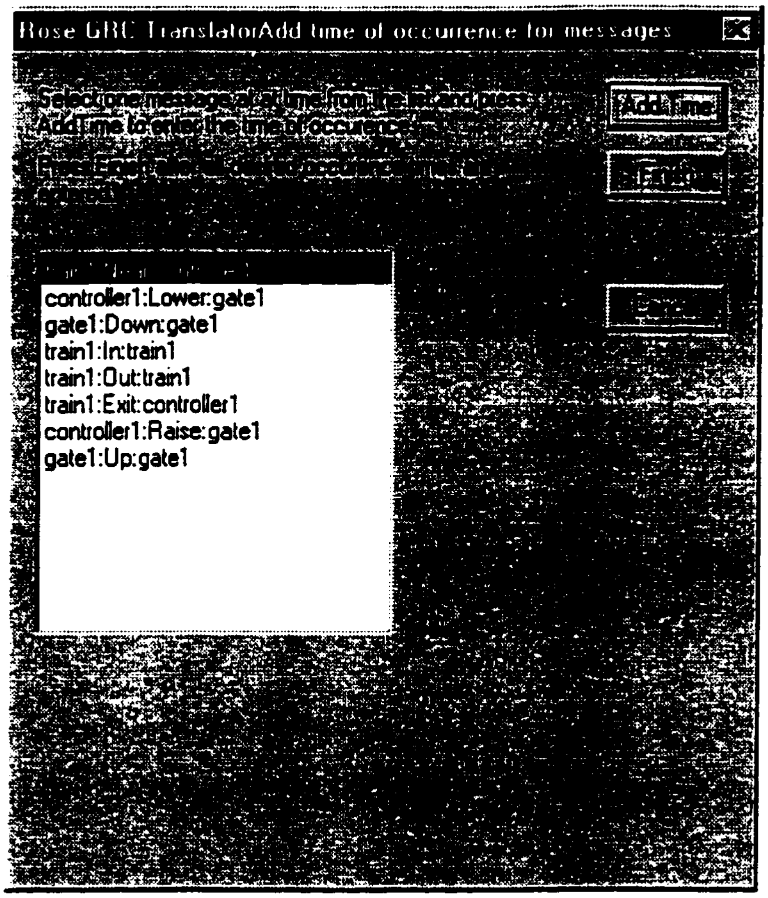
Figure 45: Dialog Box for entering subsystem name

Dialog Box for Message Times

A set of dialog boxes is used for allowing the user to enter the time of occurrence of all the messages in a sequence diagram. The main dialog box, illustrated in Figure 46 has the title "Rose-GRC translator - Add time of occurrence for messages". There is a text with instructions for the user, followed by a list composed of all messages extracted from a sequence diagram. Each entry in the list is of the format

SendingObject : MessageName : ReceivingObject.

There are three buttons: AddTime, Finish and Cancel. If the Finish button is pressed, execution of the program continues. If the Cancel button is pressed, a message is given to the user and the program is terminated. If the AddTime button is pressed when a message entry is selected, a second dialog box, illustrated in Figure 47. is displayed, with title "Rose-GRC translator - Add time for message". This dialog box contains a text with instructions for the user. a text box. an OK button and a Cancel button. The text box contains by default the message entry selected in the previous dialog box. If the time is entered correctly, then the second dialog box is removed and the message entry selected is replaced by the new message entry. If the time is entered incorrectly, the second dialog box is redisplayed. The main dialog box remains displayed until the Finish or Cancel button are pressed.


Rose GRC Translator:Add time of occurrence for messages

[illegible]

controller1:Lower:gate1
gate1:Down:gate1
train1:In:train1
train1:Out:train1
train1:Exit:controller1
controller1:Raise:gate1
gate1:Up:gate1

Figure 46: Dialog box for List of messages from a sequence diagram

Dialog box for saving file name

For file saving, a pre-defined save file dialog box is used, shown in Figure 48. Within this dialog box, the user has the capability to browse the directory structure, select the complete path and file name. The file type is pre-defined to text file. The dialog

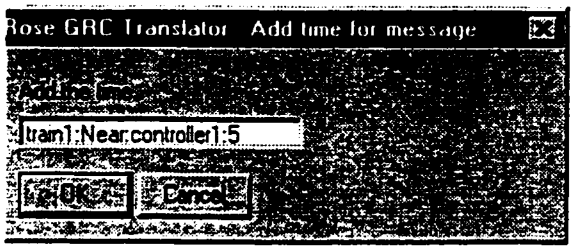

Rose GRC Translator   Add time for message
train1:Near:controller1:5
OK
Cancel

Figure 47: Dialog Box for entering the time of occurence for one message

box has two buttons: an OK button and a Cancel button. If the user presses the OK button, the file is saved and execution of the program continues. If the user presses the Cancel button, a message box is displayed indicating that Cancel was pressed and that the program ends. The program is terminated. A save file dialog box is displayed when saving the following files: the file with generic class specifications. the file with the subsystem configuration specification and the file with the message sequencing.

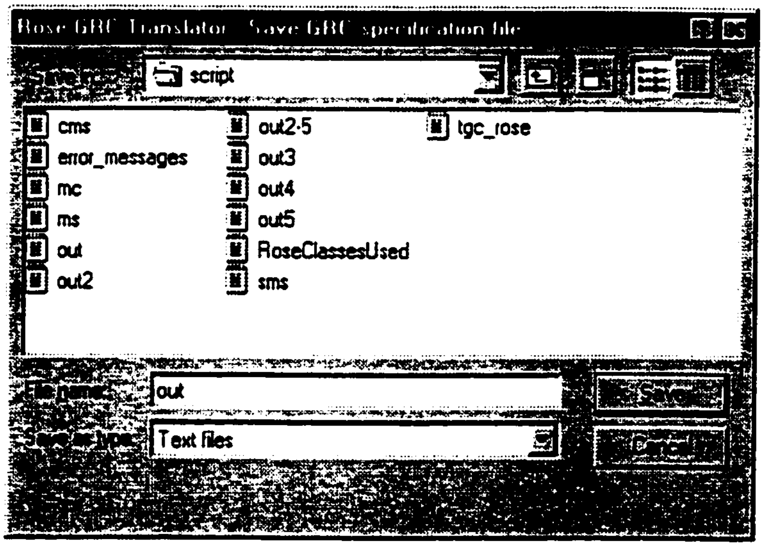


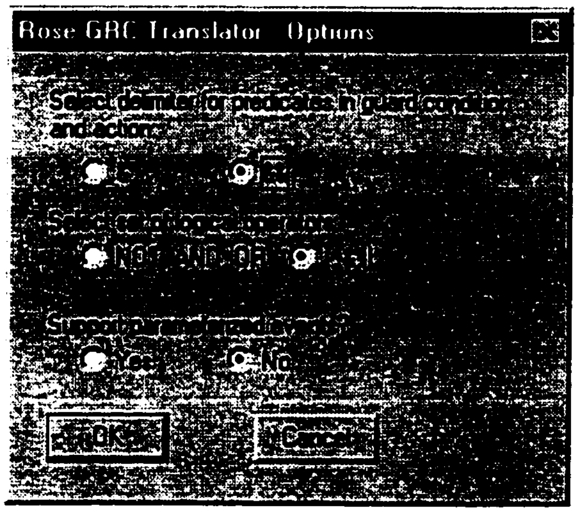
Figure 48: File save dialog for the GRC specification file

Dialog box for option selection

We define a new dialog box (Figure 49) to allow the user to select some options before running the translator. The dialog box contains text fields, option groups with option buttons, an OK button and a cancel button. There are three option groups: for selection of the delimiter string in guard condition and action, for selection of the set of logical operators allowed and for selecting parameterized events. If the user presses the OK button, execution of the program continues. If the user presses the Cancel button, the program is terminated.

The option for selection of the delimiter string for guard condition and action of a transition and the option for selection of the set of logical operators are needed for backward compatibility with older versions of the TROMLAB interpreter.

The option for parameterized events is for a future enhancement. If the output is going to be passed to TROMLAB, the option for parameterized event must not be enabled.



Rose GRC Translator Options

Select delimiter for predicates in guard condition and action

[illegible] [illegible]

Select set of logical operators

NOT AND OR [illegible]

Support parameterized events?

Yes No

Figure 49: Dialog box for option selection

5.4.4 Utility Functions

RoseScript provides functions such as Item$, ItemCount to facilitate parsing.

Item$ returns a string containing a list of all the items in a range from the given formatted text list. The items in a text list may be delimited by a specified set of delimiters. However, each of the delimiters can be only one character. A similar function was developed for the case when the specified delimiter contains several characters.

ItemCount returns the number of items from a given formatted text list. The items in a text list may be delimited by a specified set of delimiters. However, each of the delimiters can be only one character. A similar function was developed for the case when the specified delimiter has several characters.

5.5 Error Handling

The Rose-GRC Translator is designed to detect errors related to model consistency and correctness. The errors are reported to the user interface through message boxes. Error messages are also displayed in the log window. (See Section 5.4.3).

5.5.1 Error Handling in the Translator

The user is given the option of pressing the Cancel button in the message box to stop the translation program right after an error is detected and reported. or to continue until the entire system is checked, by pressing the OK button for each error. With the second option, the user can view all the errors in the error log window and/or save the log window to a file.

The Rose-GRC Translator checks for the following errors. and returns the message texts in Figures 50 and 51.

Class diagram

- All classes in a class diagram must have stereotype ≪GRC≫ or ≪PortType≫.

Generic class

- Each GRC must have at least one port type.
- Each GRC must have a state machine.

Attributes

- All attributes of a GRC class must have stereotype ≪PortType≫ or ≪DataType≫.
- PortType Attributes must have a defined port type class as type.

Port Types

- A port type class name must start with the character @.
- Port type classes must aggregate to a GRC.
- A port type class must have only one attribute, named events, of type Set, with initial value enclosed in curly brackets, representing the event set.
- Every port type must have an event list, with a direction for each event.
- Every event in the event list must be defined in the state machine of the class.
- A port type class must not have a state machine.

Transition Specifications

- Every transition must have a triggering event.
- Guard condition must have format: port-condition && enabling-condition && time-constraint-condition.
- Transition actions must have the format post-condition && time-initialization.

States

- Each GRC must have an initial state defined for each context.
- There must be one and only one transition from the Rose initial state.

Time constraints

- Time constraint variables must be named *TCvarN*, with N being a numeral.
- Time constraint conditions may have logical operators: $<$, $\leq$, $>$. $\geq$
- For each time constraint, lower-bound must be lower than upper-bound.
- Each time-constraint variable can only be used for one time-constraint.
- Each time-constraint variable must be initialized once, to 0.
- The only action on entry allowed in a state is of the form $TCvarN = -1$

Collaboration Diagram

- All objects must have a class specified.

Ports

- Port object names must start with the character @.

5.5.2 Error Handling in the GUI

- Whenever the user presses the Cancel button in a dialog box, one of the messages listed in Figure 52 may be displayed in the error message box.

- If an unacceptable combination of options was selected in the option dialog box, the message listed in figure 53 is displayed in the message error box.

5.5.3 Size Limitation Error Handling

Some of the structures in the Rose-GRC Translator contain fixed-sized arrays, which is a restriction imposed by RoseScript. Therefore, the translator has to ensure that no attempt is made to go over the fixed bounds. When such errors are detected, the error is reported and the program is terminated immediately, regardless of whether the user presses OK or Cancel button. The reason for immediate termination is that the generated specifications would be incomplete otherwise. If this occurs, the user may attempt to redesign the classes in order to decrease the number of components that caused the error. Alternatively, if the source script is available, an attempt can be made to increase the fixed size and recompile the script. See Figure 54 for a list of error messages.

```
The class <class name> is not a GRC or a PortType.
No state machine for GRC <class name>.
Association from <class name> to <class name> not supported.
PortType class <class name> has a state machine.
PortType class <class name> does not have only one attribute named events.
Port type for GRC <attribute name> does not start with @.
Port type <class name> is not an aggregation to <class name>.
No port types specified for GRC <class name>.
Attribute <attribute name> of PortType class <class name> is not named events.
Attribute <attribute name> of PortType class <class name> is not of type Set.
Initial value of attribute events is not enclosed in curly braces .
No trigger event specified for transition <transition name> for GRC <class name>.
No event list was specified for port type <class name>.
No event direction was specified for <class name> event <event name>.
<event name> is not a valid event name for GRC <class name>.
Class: <class name> attribute <attribute name>: port type <attribute type> ..
   ..does not exist.
Class: <class name> attribute <attribute name>: type <attribute type> not ..
   .. supported.
Class: <class name> attribute <attribute name>: stereotype < attribute ..
   ..stereotype> not supported.
Class <class name> is missing transition from Rose initial <state name>.
Class <class name> has too many transitions from initial state <state name>.
Class <class name> does not have an initial state or substate defined.
No event specified for transition <transition name> for GRC <class name>.
Guard condition <string> is not of the form: port-cond && enabling-cond && ..
   .. time-constraint-condition.
Action <action name> is not of the form: post-condition && time-initialization ..
   ..for GRC <class name>.
Format error in post-condition <string> for GRC <class name>.
Time constraint condition in <string> does not contain valid time constraint ..
   ..variable with name TCvar#.
Time constraint condition in <string> does not contain valid logical operator.
Time constraint condition with variable <name> has bounds inconsistency for ..
   ..GRC <class name>.
Time constraint variable <name> used more than once in GRC <class name>.
Too many actions for GRC <class name> transition action <action name>.
Format error in transition action <action name> for GRC <class name>.
Too many initializations for time constraint variable <name> in GRC <class name>.
```

Figure 50: Translator error messages

```
Wrong initialization for time constraint variable <name> in GRC <class name>.
Time constraint variable <name> in GRC <class name> is not initialized.
Action on entry <action name> of state <state name> of GRC <class name> ..
    ..is not of the form TCVar=-1.
Wrong value for time constraint variable <name> in entry action <action name> ..
    ..of state <state name> of GRC <class name>.
Object <object name> has unspecified class.
Name of port <object name> does not start with @.
Class <class name> of object <object name> is not a GRC.
Not all objects had classes specified in collaboration diagram. Specify classes ..
    .. and re-run translator.
```

Figure 51: Translator error messages—continued

```
User canceled saving the file with GRC specifications. Program ends
User canceled saving the file with SCS specifications. Program ends
User canceled saving the file with message sequence. Program ends
No collaboration diagram was selected. Program Ends.
No sequence diagram was selected. Program ends.
Selection of options was cancelled. Program ends.
Cancel was pressed. Program ends.
```

Figure 52: Error messages from GUI

```
Cannot have & as a delimiter and as a logical operator. Please make another ..
    .. selection.
```

Figure 53: Error messages from GUI for option selection

```
More than <MaxConstraints> time constraints specified for GRC <class name>.
More than <MaxStates> states specified for GRC <class name>.
More than <MaxConstraints> time constraint conditions specified for GRC ..
   .. <class name>.
More than <MaxTransitions> transitions specified for GRC <class name>.
More than <MaxEvents> events specified for GRC <class name>.
More than <MaxPortTypes> port types specified for GRC <class name>.
More than <MaxTraits> LSL traits found for GRC <class name>.
More than <MaxAttributes> attributes specified for GRC <class name>.
```

Figure 54: Error messages for size limitations

Chapter 6

Case Study: Modeling the Train-Gate-Controller Problem

6.1 Introduction

To illustrate our technique for modeling generic reactive systems in UML, we use a version of the railroad crossing problem. This problem has been discussed previously in [HL94]. [Ach95]. [AM98] as a case study to illustrate the expresivity of the TROM formalism. We focus here on Rose modeling of a generalized version of the Train-Gate-Controller system.

In this version. several trains cross a gate independently and simultaneously using non-overlapping tracks. A train may choose to cross any gate on its way. A controller controls each gate. When a train 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 it starts monitoring it. 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.

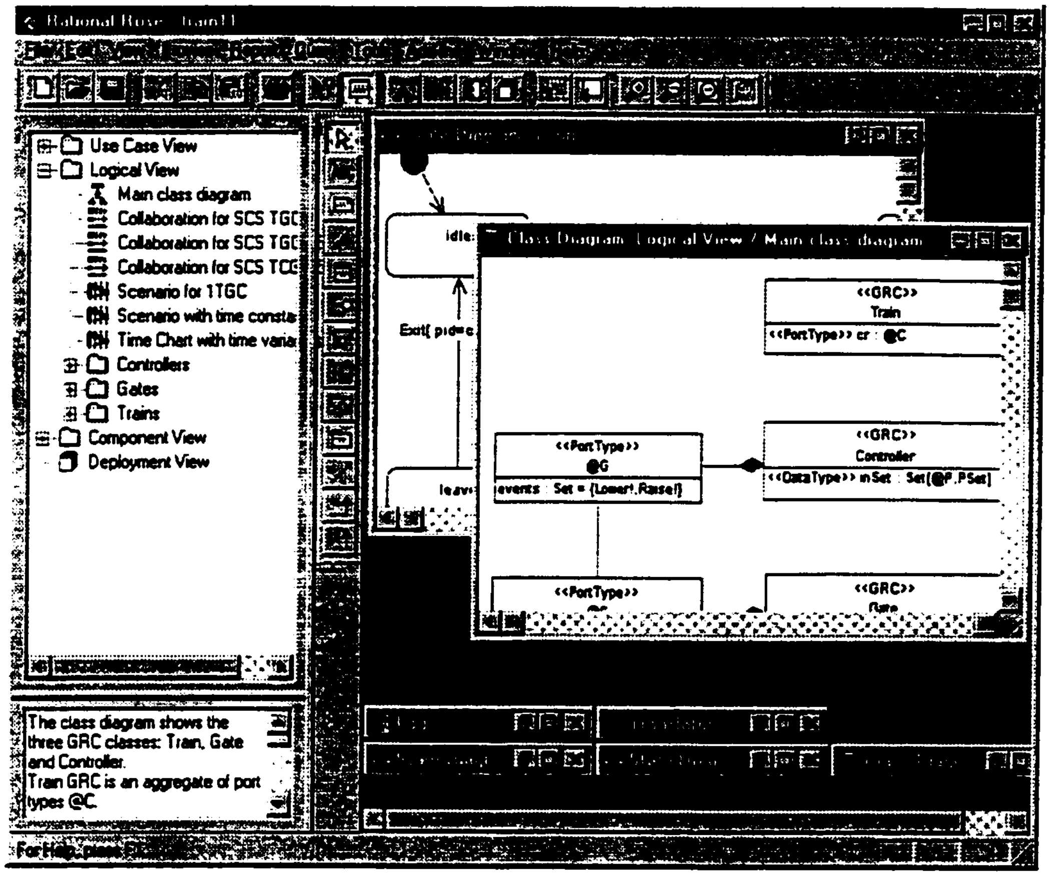
6.2 Rose model

The Rose model for the Train-Gate-Controller problem has three generic reactive classes: *Train*, *Controller*, and *Gate*. These three classes and their relationships are described in one main class diagram. Each class has a statechart diagram. The system is described in a collaboration diagram. A sequence diagram depicts a specific scenario for the system.

In our model we allow many trains to communicate with one controller, and vice versa and there is one controller for each gate.
Figure 55 shows Rose's main window.

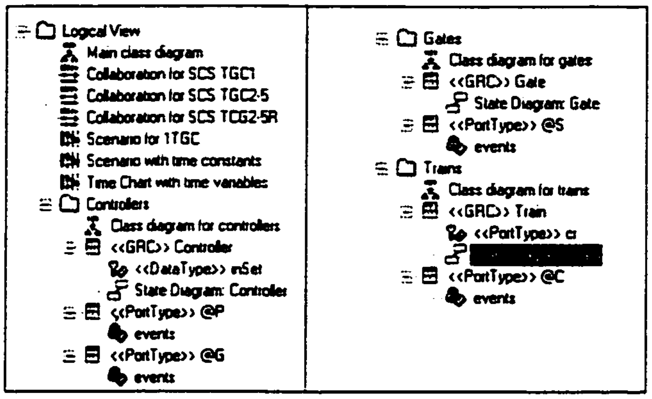
6.2.1 Overall View

The Logical View of the Rose Model for the Train-Gate-Controller has three packages. Each package holds a GRC class and its associated PortType classes. Each GRC class holds all its attributes, the state machine, and a class diagram, if it has one. The main class diagram, the collaboration and sequence diagrams belong to the main Logical View package. Figure 56 shows the Rose browser for the model's logical view.



The class diagram shows the three GRC classes: Train, Gate and Controller.
Train GRC is an aggregate of port types @C

Figure 55: Rose application with the Train-Gate-Controller model.



```
Logical View
    Main class diagram
    Collaboration for SCS TGC1
    Collaboration for SCS TGC2-5
    Collaboration for SCS TCG2-5R
    Scenario for 1TGC
    Scenario with time constants
    Time Chart with time variables
  Controllers
      Class diagram for controllers
    <<GRC>> Controller
        <<DataType>> inSet
        State Diagram: Controller
    <<PortType>> @P
        events
    <<PortType>> @G
        events
```

```
Gates
    Class diagram for gates
  <<GRC>> Gate
      State Diagram: Gate
  <<PortType>> @S
      events
Trains
    Class diagram for trains
  <<GRC>> Train
      <<PortType>> cr
      [illegible]
  <<PortType>> @C
      events
```

Figure 56: Rose browser for the logical view of the Train-Gate-Controller model.

6.2.2 Main Class Diagram

The class diagram. as depicted in Figure 57 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.*

There is an association 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.*

There is also an association 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.*

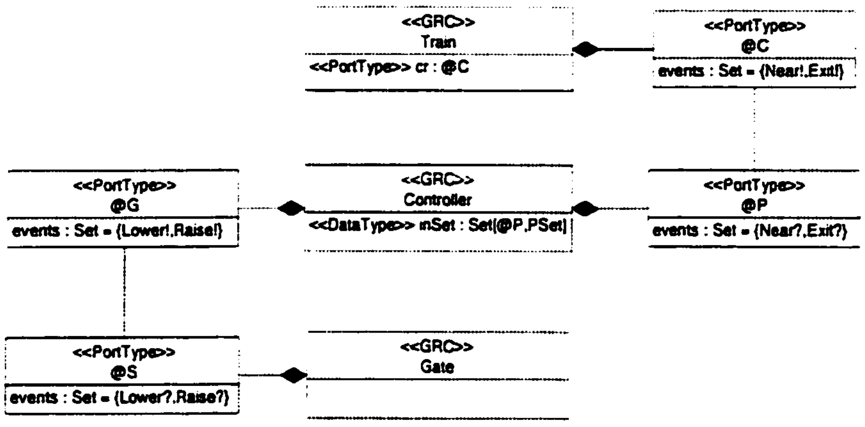
Train GRC has one port type *@C*. At port type *@C*. the following events may occur: output event *Near*. output event *Exit.*

Train GRC has one attribute. named *cr*. whose type is the port type *@C.*

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*. At port type *@S* the following events may occur: input event *Lower*, input event *Raise*.

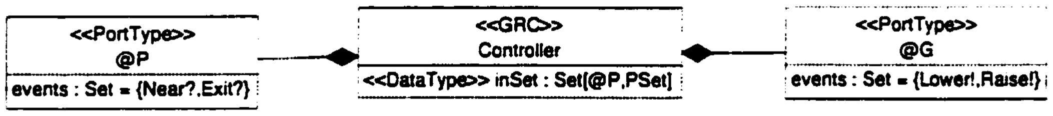


<<GRC>>
Train
<<PortType>> cr : @C
<<PortType>>
@C
events : Set = {Near!,Exit!}
<<PortType>>
@G
events : Set = {Lower!,Raise!}
<<GRC>>
Controller
<<DataType>> inSet : Set[@P,PSet]
<<PortType>>
@P
events : Set = {Near?,Exit?}
<<PortType>>
@S
events : Set = {Lower?,Raise?}
<<GRC>>
Gate

Figure 57: Main Class Diagram for Train-Gate-Controller

6.2.3 Class Diagram for One GRC

As an alternative to specifying a main class diagram with all the classes in a subsystem, the user may create a class diagram for each GRC, as shown in Figure 58 . This class diagram will contain only the GRC class and the corresponding PortType classes. To illustrate this, we show here a class diagram for the *Controller* GRC.



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

Figure 58: Class Diagram for Controller

6.2.4 Train GRC

The statechart diagram for *Train* in shown in Figure 59.

A *Train* object can be in one of four states: *idle*, *toCross*, *cross*, *leave*. *Idle* is 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 constraints, labeled *TCvar1* and *TCvar2*. *Train* goes into state *toCross*.

A transition from state *toCross* to *cross* happens when internal event *In* occurs in state *toCross*, and if the time constraint condition *TCvar1*≥2 AND *TCvar1*≤4 is true. This time constraint means that internal event *In* should occur within 2 to 4 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* happens when event *Exit* occurs in state *leave*, if the attribute *cr* has the value *pid* (pid is the identifier of the port where *Exit* occurs), and if the time constraint condition *Tcvar2*≤6 is true. This time constraint means that event *Exit* should occur within 6 time units after event *Near* occurs in state *idle*.

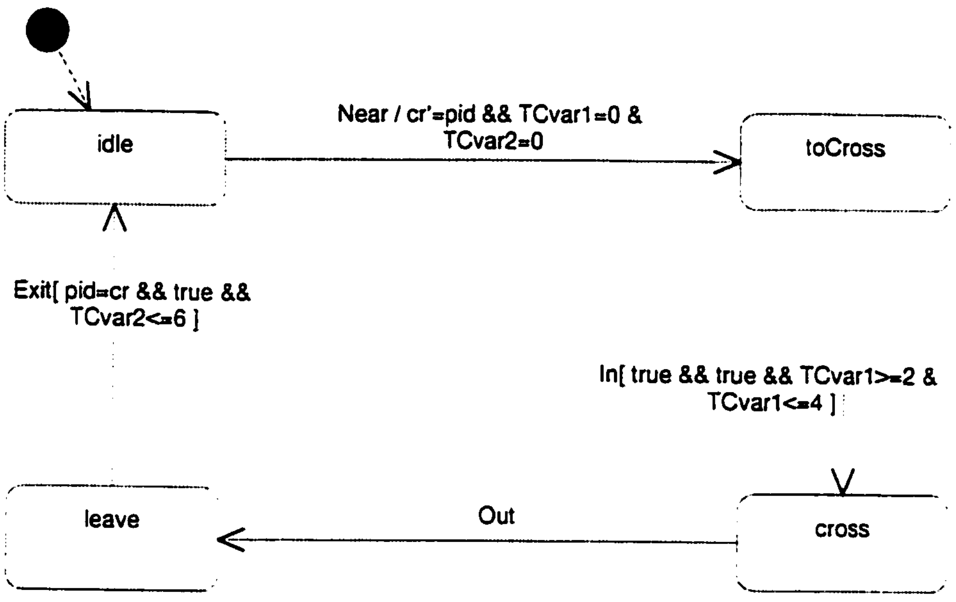

idle
Near / cr'=pid && TCvar1=0 & TCvar2=0
toCross
Exit[pid=cr && true && TCvar2<=6]
In[true && true && TCvar1>=2 & TCvar1<=4]
leave
Out
cross

Figure 59: Statechart Diagram for Train

6.2.5 Controller GRC

The statechart diagram for *Controller* is shown in Figure 60.

A *Controller* object can be in one of four states: *idle*, *activate*, *monitor*, *deactivate*. *Idle* is 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 of the port where the new event *Near* occurs). The *controller* remains in the state *activate*.

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

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

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

When event *Exit* occurs in state *monitor*, if the identifier (*pid*) of the port where event *Exit* was received is a member of *inSet* and if the size of *inSet* 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 event *Raise* should occur within 1 time unit after event *Exit* was received from the last *Train* in the crossing.

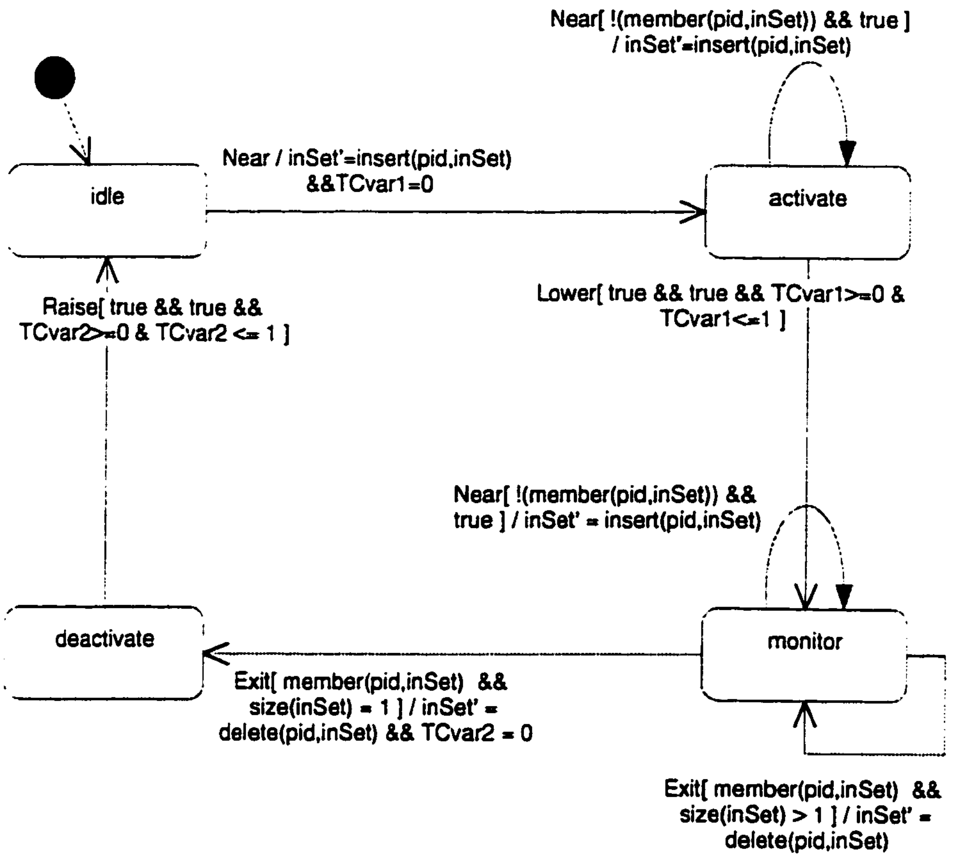

Near[!(member(pid,inSet)) && true]
/ inSet'=insert(pid,inSet)
idle
Near / inSet'=insert(pid,inSet)
&&TCvar1=0
activate
Raise[true && true &&
TCvar2>=0 & TCvar2 <= 1]
Lower[true && true && TCvar1>=0 &
TCvar1<=1]
Near[!(member(pid,inSet)) &&
true] / inSet' = insert(pid,inSet)
deactivate
monitor
Exit[member(pid,inSet)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

Figure 60: Statechart Diagram for Controller

6.2.6 Gate GRC

The statechart diagram for *Gate* in shown in Figure 61.

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

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*.

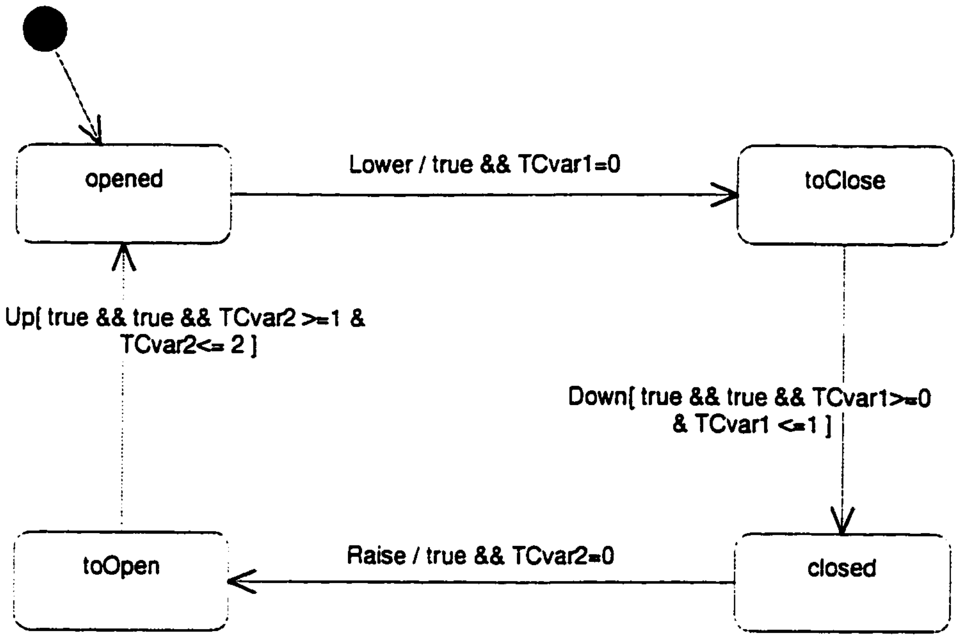

opened
Lower / true && TCvar1=0
toClose
Up[true && true && TCvar2 >=1 & TCvar2<= 2]
Down[true && true && TCvar1>=0 & TCvar1 <=1]
toOpen
Raise / true && TCvar2=0
closed

Figure 61: Statechart Diagram for Gate

6.2.7 Collaboration Diagram

A collaboration diagram depicts a system, or, for larger systems, a subsystem. First we specify a system with one train, one controller and one gate, named *TrainGate-Controller1*. Second, we specify a system with five trains, two controllers and two gates, named *TrainGateController2*.

TrainGateController1
The collaboration diagram for this subsystem is shown in Figure 62.

This system has one train object (*Train1*), one controller object (*Controller1*) and one gate object (*Gate1*).

Train1 has one port *@C1* of type *@C*, for communication with *Controller1*.

Controller1 has one port *@P1* of type *@P*, for communication with *Train1* and another port *@G1* of type *@G* for communication with *Gate1*.

Gate1 has one port *@S1* of type *@S* for communication with *Controller1*.

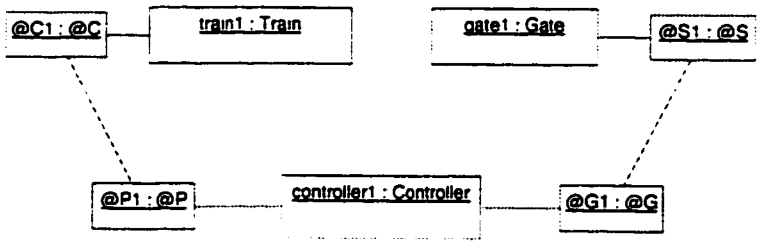


Figure 62: Collaboration diagram for subsystem TrainGateController1

TrainGateController2
The collaboration diagram for this subsystem is shown in Figure 63

This subsystem is for a particular subsystem configuration, composed of five trains, two gates and two controllers. The routes of the five trains are such that one of the trains can go through both gates, two other trains can only go through the first gate and the last two trains can only go through the second gate.

This subsystem configuration captures the fact that not all trains communicate with all controllers and, conversely, that not all controllers communicate with all trains.

6.2.8 Sequence Diagram

Figure 64 shows a sequence diagram that depicts a particular scenario in a subsystem with one train, one controller and one gate. We use sequence diagrams to provide for a future enhancement, namely for a formal verifier (see Section 8.3). Only GRC objects are shown in the scenario, without the ports, for simplicity. The messages are ordered, and sequence numbers are shown with the labels.

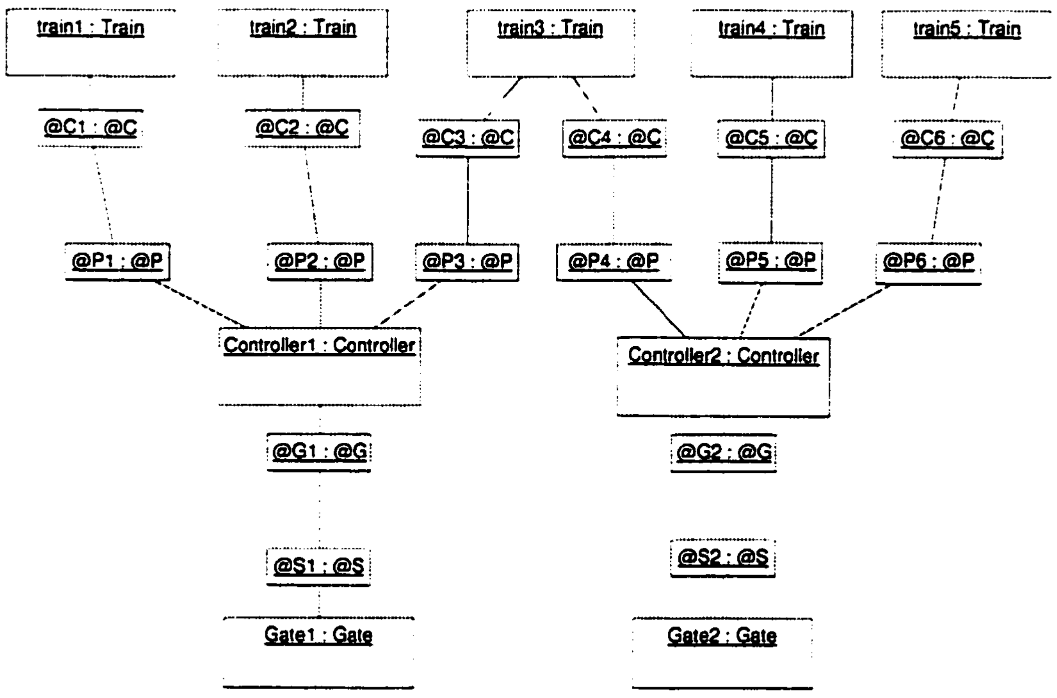


Figure 63: Collaboration diagram for subsystem TrainGateController2

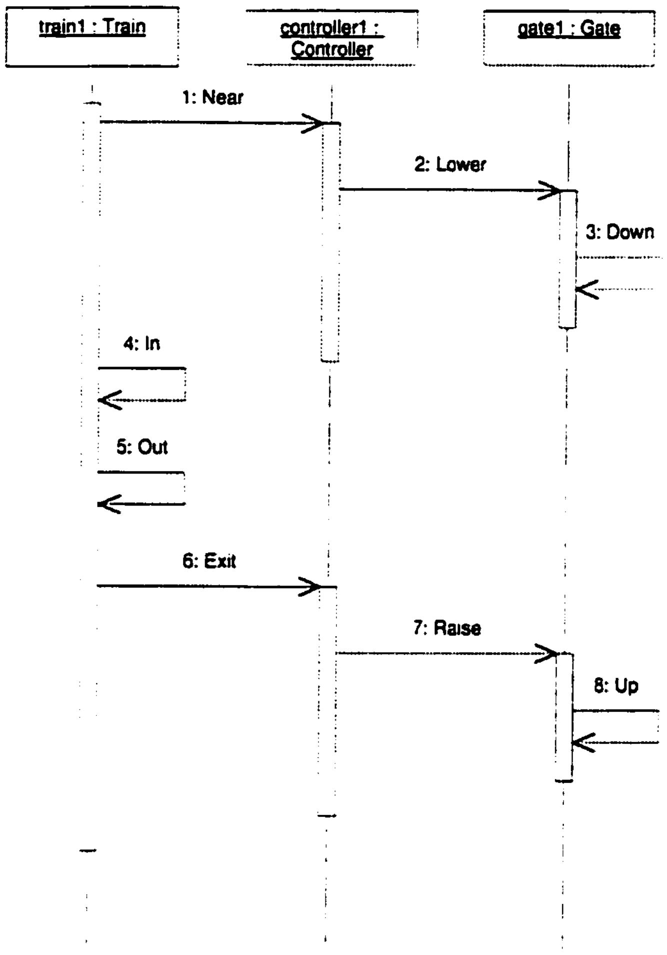


Figure 64: Sequence diagram for a scenario in subsystem TrainGateController1

6.3 Formal Model

6.3.1 GRC Specification for Train

The formal specification for GRC Train, shown in Figure 65, is generated by the translator from information in the main class diagram and the statechart diagram of GRC Train, shown in Figures 57 and 59.

```
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 65: Formal specification for GRC Train

6.3.2 GRC Specification for Controller

The formal specification for GRC Controller, shown in Figure 66, is generated by the translator from information in the main class diagram and the statechart diagram of GRC Controller, shown in Figures 57 and 60.

6.3.3 GRC Specification for Gate

The formal specification for GRC Gate, shown in Figure 67, is generated by the translator from information in the main class diagram and the statechart diagram of GRC Gate, shown in Figures 57 and 61.

6.3.4 Configuration Specification for TrainGateController1

The formal specification for subsystem TrainGateController1, shown in Figure 68, is generated by the translator from information in the collaboration diagram for subsystem TrainGateController1, shown in Figure 62.

6.3.5 Configuration Specification for TrainGateController2

The formal specification for subsystem TrainGateController2, shown in Figure 69, is generated by the translator from information in the collaboration diagram for subsystem TrainGateController2, shown in Figure 63.

6.3.6 Message Sequencing

A message sequencing for a particular scenario in subsystem TrainGateController1, shown in Figure 70 is extracted by the translator from information in the sequence diagram for a scenario in subsystem TrainGateController1, shown in Figure 64. The time of occurrence is entered by the user during translation.

6.4 Case Study Conclusion

Upon completion of this case study, we conclude that the Rose-GRC Translator:

- is capable to produce output conforming to the formally specified GRC (Section 5.2)
- is compatible with the interface of the Interpreter in TROMAB

```
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 66: Formal specification for GRC Controller

```
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 67: Formal specification for GRC Gate

```
SCS TrainGateController1
      Includes:
      Instantiate:
            gate1::Gate[@S:1];
            train1::Train[@C:1];
            controller1::Controller[@P:1, @G:1];
      Configure:
            controller1.@G1:@G — gate1.@S1:@S;
            controller1.@P1:@P — train1.@C1:@C;
end
```

Figure 68: Formal specification for subsystem TrainGateController1

```
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 69: Formal specification for subsystem TrainGateController2

```
train1:Near:controller1:5
controller1:Lower:gate1:6
gate1:Down:gate1:7
train1:In:train1:8
train1:Out:train1:9
train1:Exit:controller1:10
controller1:Raise:gate1:11
gate1:Up:gate1:13
```

Figure 70: A message sequence in subsystem TrainGateController2

Chapter 7

Industrial Application: Mobile Originating Short Message Services

7.1 Introduction

This chapter presents an application from the mobile telephony industry: Mobile Originating Short Message Services (MO SMS). The problem is described in Section 7.2. A formal model for a mobile telecommunication system was presented in [AAR95], to illustrate the use of the TROM based methodology. That model used parameterized events. However the current implementation of TROMLAB does not support parameterized events. Here we first model the MO SMS problem in Rose using parameterized events, and we use the translator to produce the formal specifications. The translator supports parameterized events to some extent, however the output produced is based on [AAR95] and not on a formal grammar. We show the Rose model for MO SMS in Section 7.3 and the corresponding formal specifications in Section 7.4. In order to produce an output compatible with the TROMLAB tools, we model the MO SMS problem again in Rose (Section 7.5), this time without the use of parameterized events, by flattening the domain of events. We then translate the Rose model into formal specifications (Section 7.6).

7.2 Problem Description

Mobile Originating Short Message Services is a teleservice provided by the Ericsson cellular system, CMS 8800, according to the North American standard, D-AMPS. A mobile station (MS) may send a short text message to a message center (MC) through the Ericsson mobile network, CMS8800. The CMS8800 network is able to receive the short message and transparently forward it to the appropriate message center and to forward the reply from the message center back to the mobile. The final destination of the short message could be another mobile subscriber or a land-line subscriber, but is irrelevant to the scope of the problem.

A mobile station sends the short message through a digital control channel (DCCH), in the R-DATA message, as specified in [TIA98]. After performing a number of checks, the CMS8800 network forwards the short message to the *MC* with the message ShortMessageDeliveryPointToPoint(SMDPP) Invoke, as specified in [TIA97]. The *MC* acknowledges reception of the short message by sending SMDPP Return Result to the CMS8800 network. The CMS8800 network sends R-DATA ACCEPT or R-DATA REJECT to the mobile station depending on whether the short message transmission was successful or not. The CMS8800 network may send REORDER to the MS in the case when the format of the mobile station identification is not acceptable.

Figure 71 shows the use case diagram for MO SMS.

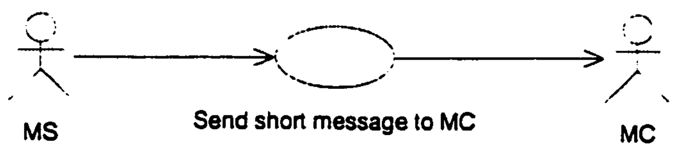
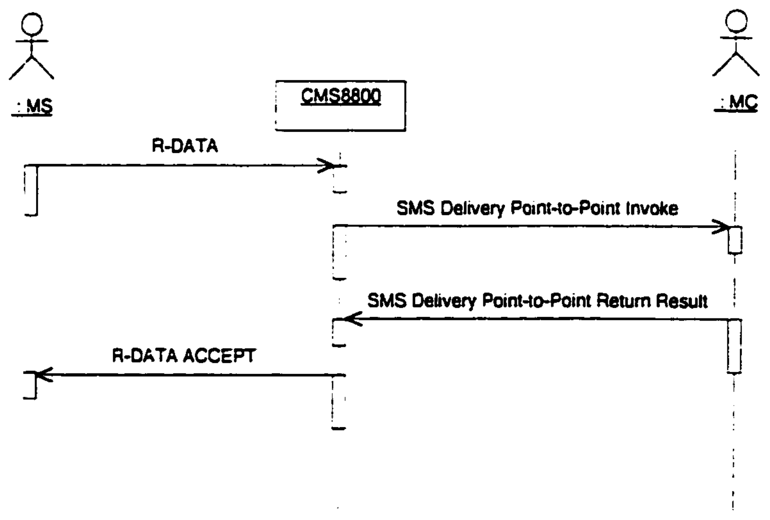


Figure 71: Use case diagram for MO SMS

Figure 72 [Eri98] shows the main flow of messages between the *MS*, the CMS8800 network and the *MC*, in the case when a short message is successfully forwarded to the *MC*.

Figure 73 [Eri98] shows the flow of messages between the *MS* and the CMS8800 network, in the case when a short message cannot be forwarded to the *MC* due to an unacceptable format in the mobile station identification.



: MS
CMS8800
: MC
R-DATA
SMS Delivery Point-to-Point Invoke
SMS Delivery Point-to-Point Return Result
R-DATA ACCEPT

Figure 72: Main flow of messages for a successful MO SMS

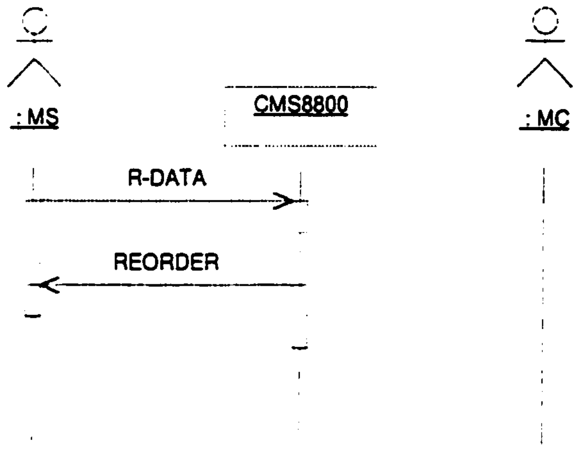


Figure 73: Flow of messages for a MO SMS access with invalid MSID

Figure 74 [Eri98] shows the flow of messages between the *MS* and the CMS8800 network, in the case when a short message cannot be forwarded to the *MC* due to length of short message or to an invalid subscriber identification or to an invalid destination address.

Figure 75 [Eri98] shows the flow of messages between the *MS*, the CMS8800 network and the *MC*, in the case when a short message cannot be stored in the *MC*.

The main timing requirement for MO SMS is the following: If the mobile station does not receive an acknowledgment after a certain time (20 super-frames), then it

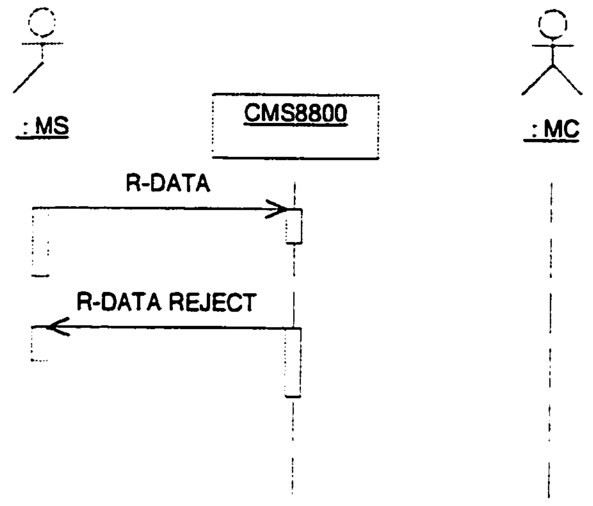
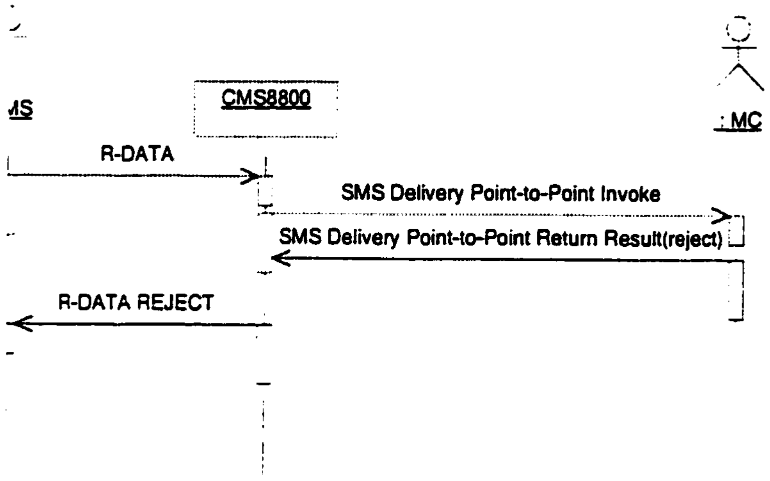


Figure 74: Flow of messages for a MO SMS access that cannot be forwarded to MC



MS
CMS8800
: MC
R-DATA
SMS Delivery Point-to-Point Invoke
SMS Delivery Point-to-Point Return Result(reject)
R-DATA REJECT

Figure 75: Flow of messages for a MO SMS access that cannot be stored in MC

will attempt to send the message once more.

We will model the mobile, the CMS8800 network and the message center as generic reactive classes, and we will specify their behavior with respect to MO SMS functionality only, with focus on the CMS8800 network.

Mobile Station

With respect to MO SMS functionality, the initial state for the *MS* is the *DCCH Camping state* [TIA98]. From the *DCCH Camping* state, the *MS* sends R-DATA to the CMS8800 network and goes to a waiting state.

In this waiting state the *MS* may receive one of these messages: R-DATAACCEPT, R-DATAREJECT and REORDER. If R-DATAACCEPT is received, the *MS* must ensure if the message is intended for it. If R-DATAREJECT is received, the *MS* must ensure if the message is intended for it. If REORDER is received, the *MS* must ensure if the message is intended for it and then it may re-send the R-DATA with the same data, using the alternate *MSID*, if available, and it goes to a waiting state. Otherwise it returns to the *DCCH Camping* state.

If a timeout occurs (no message received from the CMS8800 network after 20 super-frames), if it is the first timeout, the *MS* resends the R-DATA message and goes into a waiting state. Otherwise it returns to the *DCCH Camping* state. If the timeout occurs after the *MS* has resent the R-DATA message due to a REORDER, then it should not attempt to re-send the R-DATA again.

Message Center

As stated before, in this model we focus on the CMS8800 network. From its perspective, after the SMDPP Invoke message is sent to the message center, the conditions under which the message can or cannot be stored in the message center are irrelevant. It is important to distinguish between an SMDPP Return Result message indicating success and one indicating failure sent from the message center to the CMS8800 network.

CMS8800 Network

In order to support MO SMS functionality on DCCH, the CMS 8800 network shall perform the following functions:

- Receive R-DATA message from the *MS*.
- Check MO SMS feature activation.
- Check MSID format and, if necessary, send a REORDER to the *MS*.
- Check R-DATA message length.
- Identify the subscriber.
- Handle multiple MO SMS accesses in the case when a timeout occurs in the *MS*.
- Send SMDPP Invoke to the *MC*.

- Receive SMDPP Return Result from the *MC*.
- Send R-DATA ACCEPT to the *MS*.
- Send R-DATA REJECT to the *MS*.
- If a timeout occurs when the CMS8800 waits for a reply from the *MC*. no reply should be sent to the *MS* in order to allow it to timeout.

R-DATA parameters
At this level of abstraction, the following parameters are relevant in the R-DATA message: Bearer data length, Subscriber identifier. MSID, Destination Address. Based on the value of these parameters. the CMS8800 will go into different states. The other parameters. such as the text of the short message, the originating address and subaddress, and the destination subaddress may be considered implicit because their value is not checked at this level of abstraction.

7.3 Rose Model with Parameterized Events

7.3.1 Class diagram

We model this problem with three generic reactive classes: MS. CMS8800 and MC. and we create a class diagram (Figure 76) for the three GRCs and their respective port types.

The *MS* class has a port type @C for communication with the *CMS8800* class.

The CMS8800 class has a port type @MO for communication with a MS. and another port type @S for communication with a MC.

The MC class has a port type @M for communication with the CMS8800 class.

The attribute *events* from each port type class has the initial value equal to the set of input and output events that may occur at a port of that type. In order to simplify the diagram. we selected Rose's option to "Not show attributes" for all the port type classes.

The MS class has the following attributes:

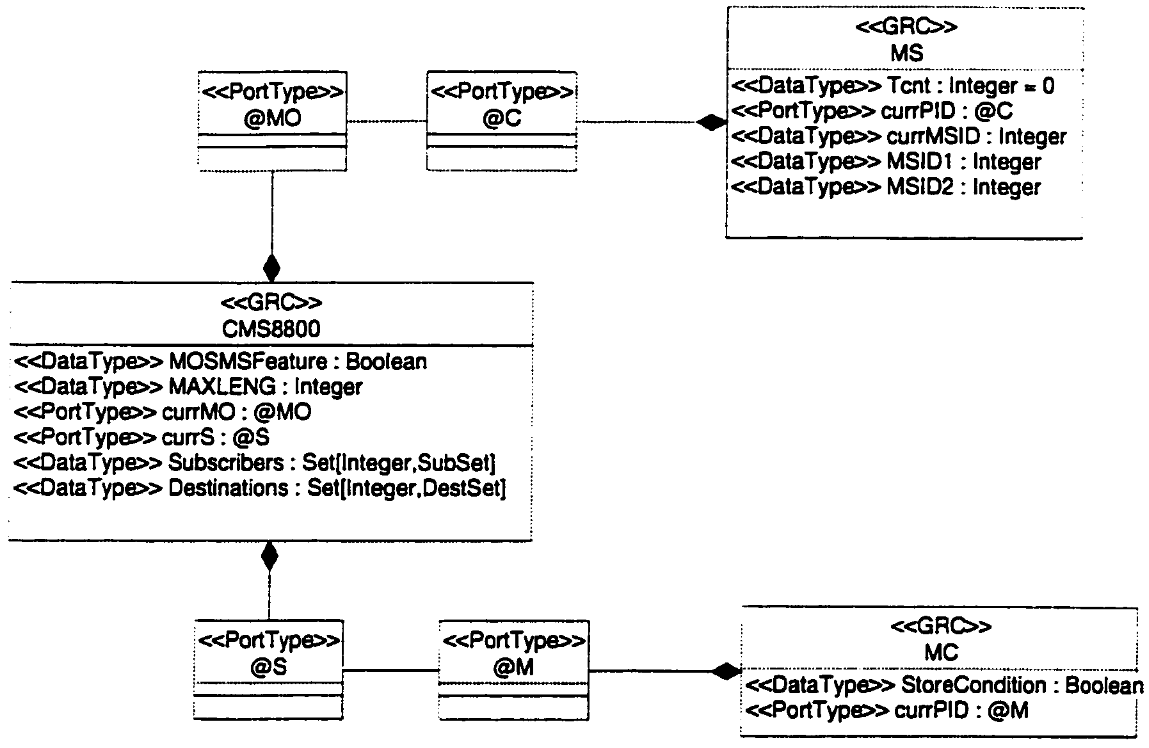
- Tcnt is a counter for timeouts, initialized to 0.
- currPID stores the pid of the current port of type @C.
- currMSID stores the current mobile station identification.
- MSID1 is the primary mobile station identification.
- MSID2 is the alternate (optional) mobile station identification.

The CMS8800 class has the following attributes:

- MOSMSFeature is a boolean that indicates if the CMS8800 network supports the MO SMS functionality. Setting this value is outside the scope of this MO SMS model.
- MAXLENG stores the maximum accepted length of a short message accepted by the CMS8800 network. Setting this value is outside the scope of this MO SMS model.
- currMO stores the pid of the current port of type @MO.
- currS stores the pid of the current port of type @S.
- Subscribers is a set of integer elements that stores the subscriber identifications (SubID) of all subscribers that are permitted to make a MO SMS access.
- Destinations is a set of integer elements that stores the destination addresses (DestAddr) of all valid destinations. This is used also to obtain the port of the appropriate message center.

The MC class has the following attributes:

- StoreCondition is a boolean indicating whether the short message may or may not be stored in the message center.
- currPID stores the pid of the current port of type @M.



<<PortType>>
@MO
<<PortType>>
@C
<<GRC>>
MS
<<DataType>> Tcnt : Integer = 0
<<PortType>> currPID : @C
<<DataType>> currMSID : Integer
<<DataType>> MSID1 : Integer
<<DataType>> MSID2 : Integer
<<GRC>>
CMS8800
<<DataType>> MOSMSFeature : Boolean
<<DataType>> MAXLENG : Integer
<<PortType>> currMO : @MO
<<PortType>> currS : @S
<<DataType>> Subscribers : Set[Integer,SubSet]
<<DataType>> Destinations : Set[Integer,DestSet]
<<PortType>>
@S
<<PortType>>
@M
<<GRC>>
MC
<<DataType>> StoreCondition : Boolean
<<PortType>> currPID : @M

Figure 76: Class diagram with all GRCs for MO SMS

7.3.2 MS GRC

Figure 77 illustrates the statechart diagram for the MS GRC.

When the *MS* sends a R-DATA event in state *DCCHCamping*, it goes in state *MOSMSproceeding*. If R-DATAACCEPT or R-DATAREJECT is received in this state, the *MS* returns to *DCCHCamping* after releasing its resources. If REORDER is received in this state and an alternate MSID is available, the *MS* sends again R-DATA and goes into state *waitresend*. If R-DATAACCEPT or R-DATAREJECT is received in this state, the *MS* returns to *DCCHCamping* after releasing its resources. If REORDER is received in this state, the *MS* treats it as a reject. If a timeout occurs in state *waitresend*, the *MS* returns to *DCCHCamping*. If a timeout occurs in state *MOSMSproceeding*, the *MS* resends R-DATA and goes into state *wait2*. If R-DATAACCEPT or R-DATAREJECT is received in this state, the *MS* returns to *DCCHCamping* after releasing its resources. If REORDER is received in this state and an alternate MSID is available, the *MS* sends again R-DATA and goes into state *waitresend*. If a timeout occurs in state *wait2*, the *MS* returns to *DCCHCamping*.

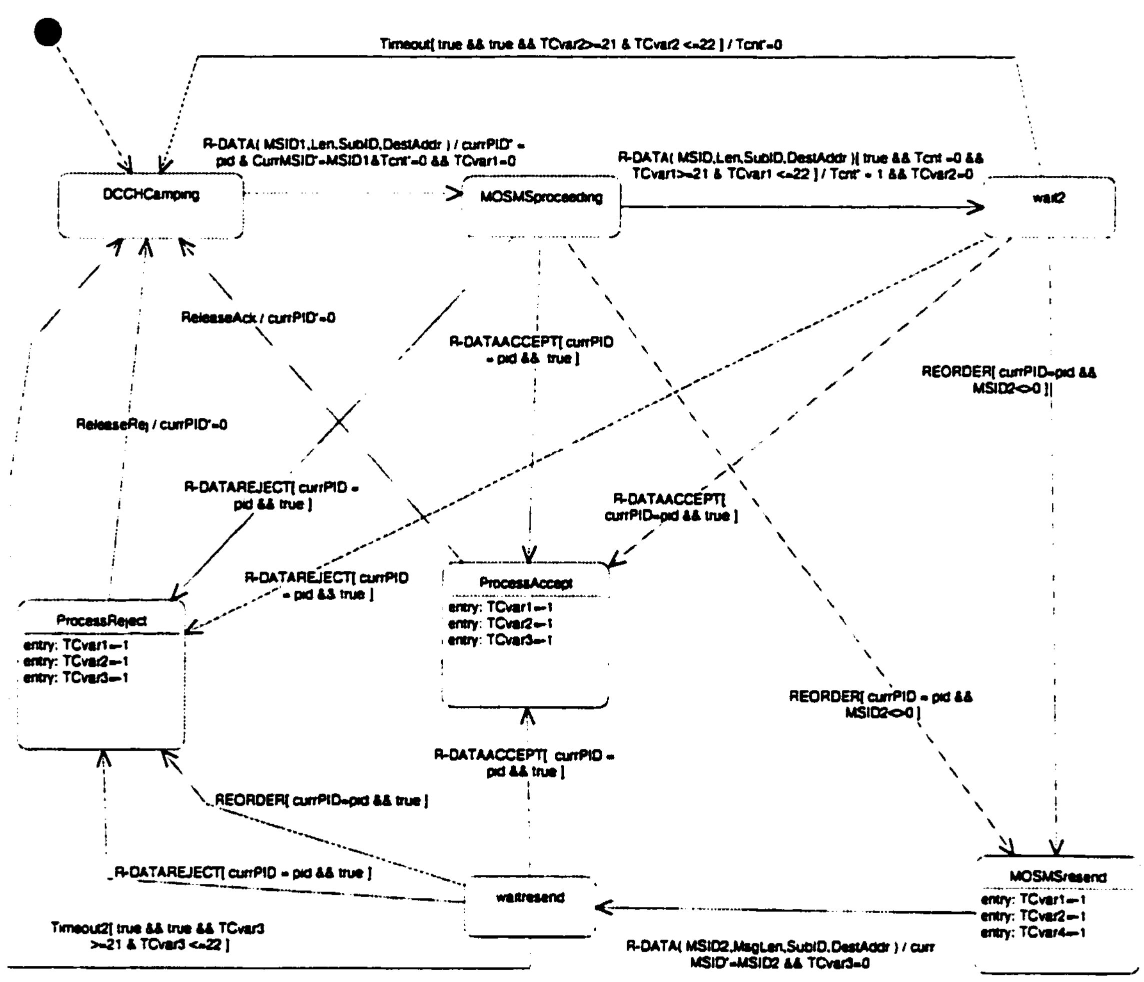

Timeout[true && true && TCvar2>=21 & TCvar2 <=22] / Tcnt'=0
R-DATA(MSID1,Len,SubID,DestAddr) / currPID' = pid & CurrMSID'=MSID1&Tcnt'=0 && TCvar1=0
R-DATA(MSID,Len,SubID,DestAddr)[true && Tcnt =0 && TCvar1>=21 & TCvar1 <=22] / Tcnt' = 1 && TCvar2=0
DCCHCamping
MOSMSproceeding
wait2
ReleaseAck / currPID'=0
R-DATAACCEPT[currPID = pid && true]
REORDER[currPID=pid && MSID2<>0]|
ReleaseReq / currPID'=0
R-DATAREJECT[currPID = pid && true]
R-DATAACCEPT[currPID=pid && true]
R-DATAREJECT[currPID = pid && true]
ProcessAccept
entry: TCvar1=-1
entry: TCvar2=-1
entry: TCvar3=-1
ProcessReject
entry: TCvar1=-1
entry: TCvar2=-1
entry: TCvar3=-1
REORDER[currPID = pid && MSID2<>0]
R-DATAACCEPT[currPID = pid && true]
REORDER[currPID=pid && true]
R-DATAREJECT[currPID = pid && true]
waitresend
MOSMSresend
entry: TCvar1=-1
entry: TCvar2=-1
entry: TCvar4=-1
Timeout2[true && true && TCvar3 >=21 & TCvar3 <=22]
R-DATA(MSID2,MsgLen,SubID,DestAddr) / curr MSID'=MSID2 && TCvar3=0

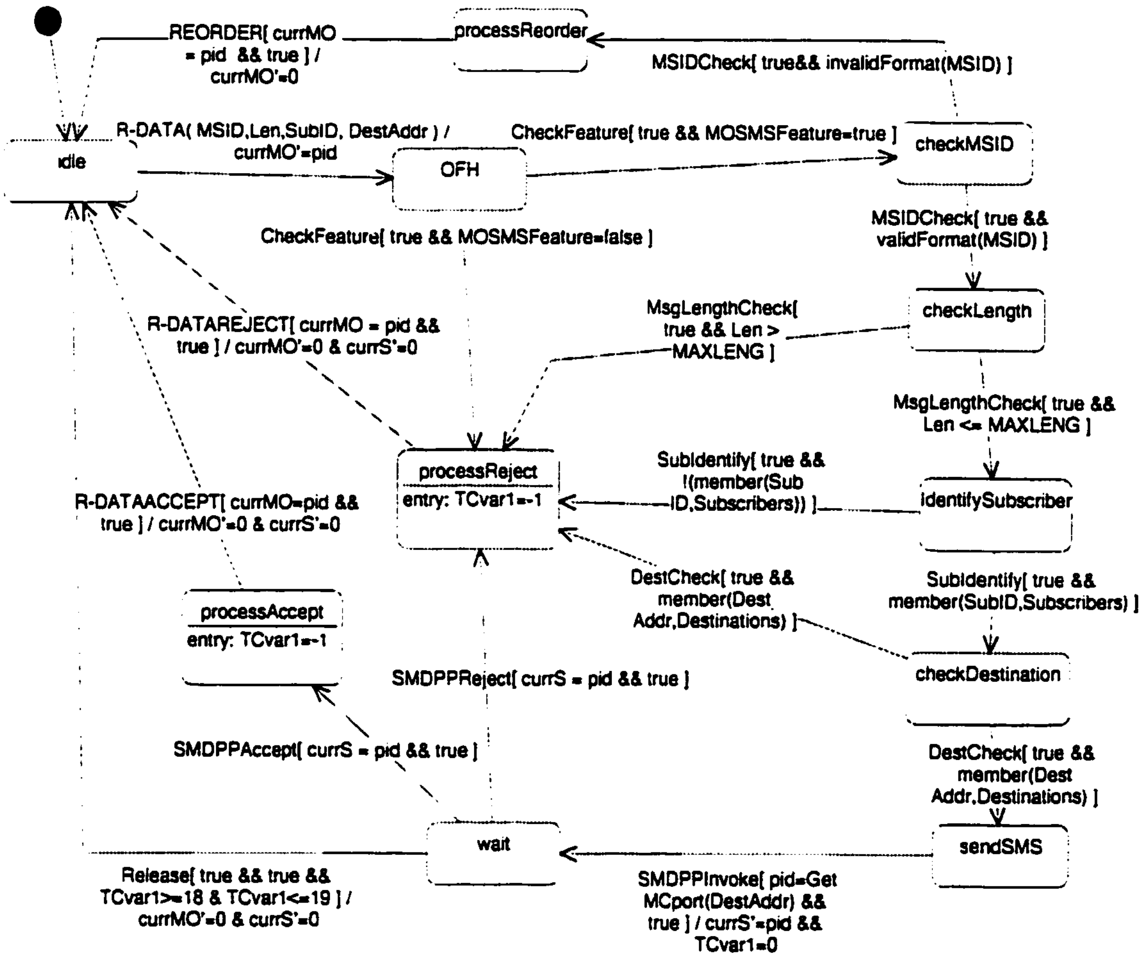
Figure 77: Statechart diagram for MS GRC

7.3.3 CMS8800 GRC

Figure 78 illustrates the statechart diagram for the CMS8800 GRC.

When the *CMS8800* receives an R-DATA, it checks if the MOSMS feature is supported. If the MOSMS feature is not supported, *CMS8800* sends R-DATAREJECT and returns to *idle*. If the MOSMS feature is supported, *CMS8800* checks the format of the MSID received. If the MSID format is invalid, it sends REORDER and returns to *idle*. If the MSID format is valid, *CMS8800* checks if the message length received is less than the maximum accepted. If it is longer than accepted, *CMS8800* sends R-DATAREJECT and returns to *idle*. If the message length is acceptable, *CMS8800* checks the identity of the subscriber. If the subscriber is not identified, *CMS8800* sends R-DATAREJECT and returns to *idle*. If the subscriber is identified, *CMS8800* checks the destination address. If the destination address is invalid, *CMS8800* sends

R-DATAREJECT and returns to *idle*. If the destination is valid, *CMS8800* retrieves the port of the *MC* corresponding to the destination address and sends SMDPPInvoke to that *MC*. If *CMS8800* receives a SMDPPAccept, it sends R-DATAACCEPT and returns to *idle*. If *CMS8800* receives a SMDPPReject, it sends R-DATAREJECT and returns to *idle*. If *CMS8800* does not receive anything, whithin 18 to 19 time units, it returns to *idle*.



REORDER[currMO = pid && true] / currMO'=0
processReorder
MSIDCheck[true&& invalidFormat(MSID)]
R-DATA(MSID,Len,SubID, DestAddr) / currMO'=pid
idle
OFH
CheckFeature[true && MOSMSFeature=true]
checkMSID
CheckFeature[true && MOSMSFeature=false]
MSIDCheck[true && validFormat(MSID)]
R-DATAREJECT[currMO = pid && true] / currMO'=0 & currS'=0
MsgLengthCheck[true && Len > MAXLENG]
checkLength
MsgLengthCheck[true && Len <= MAXLENG]
SubIdentify[true && !(member(SubID,Subscribers))]
processReject
entry: TCvar1=-1
identifySubscriber
R-DATAACCEPT[currMO=pid && true] / currMO'=0 & currS'=0
DestCheck[true && member(DestAddr,Destinations)]
SubIdentify[true && member(SubID,Subscribers)]
processAccept
entry: TCvar1=-1
SMDPPReject[currS = pid && true]
checkDestination
SMDPPAccept[currS = pid && true]
DestCheck[true && member(DestAddr,Destinations)]
wait
sendSMS
Release[true && true && TCvar1>=18 & TCvar1<=19] / currMO'=0 & currS'=0
SMDPPInvoke[pid=GetMCport(DestAddr) && true] / currS'=pid && TCvar1=0

Figure 78: Statechart diagram for CMS8800 GRC

7.3.4 MC GRC

Figure 79 illustrates the statechart diagram for the MC GRC.

When the *MC* receives SMDPPInvoke, it checks if the short message can be stored. If it can, than it is stored and *MC* sends SMDPPAccept and returns to *idle*. If the short message cannot be stored, *MC* sends SMDPPReject and returns to *idle*.

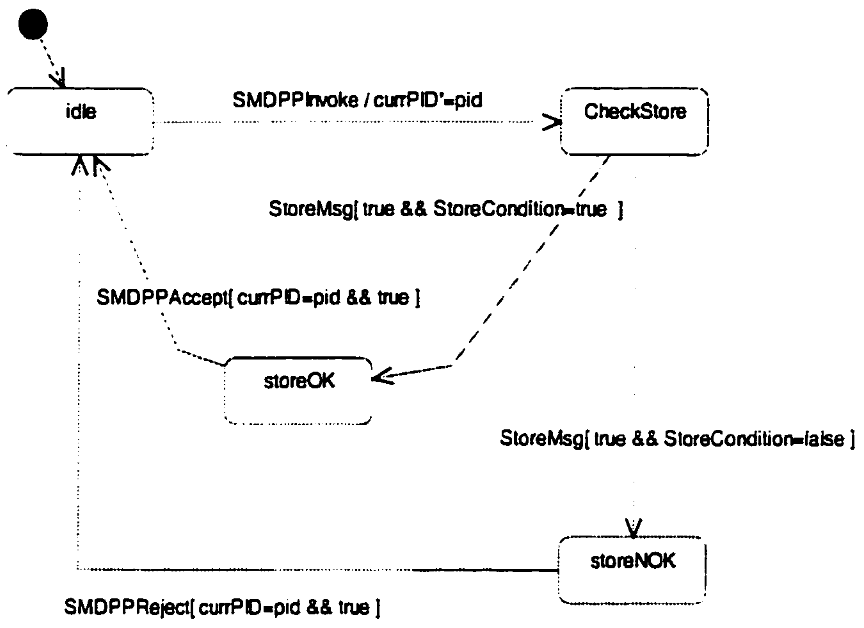

idle
SMDPPInvoke / currPID'=pid
CheckStore
StoreMsg[true && StoreCondition=true]
SMDPPAccept[currPID=pid && true]
storeOK
StoreMsg[true && StoreCondition=false]
storeNOK
SMDPPReject[currPID=pid && true]

Figure 79: Statechart diagram for MC GRC

7.3.5 A subsystem

Figure 80 illustrates the collaboration diagram for a subsystem with one MS, one CMS8800 network and one MC.

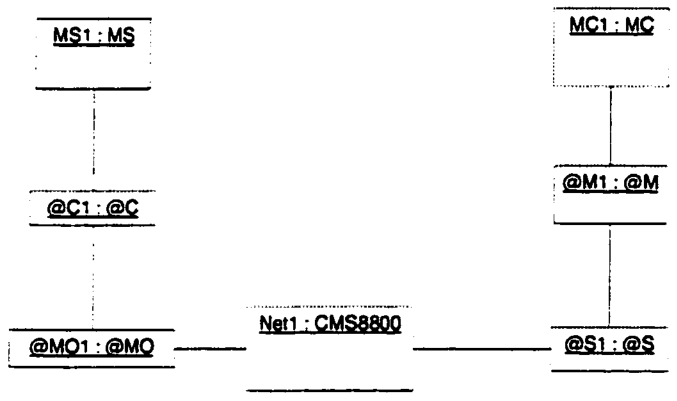


Figure 80: Collaboration diagram for a subsystem

7.4 Formal Model with Parameterized Events

7.4.1 The MC class

Figure 81 shows the formal specification for the MC GRC.

```
Class MC [@M]
Events: SMDPPInvoke?@M, StoreMsg, SMDPPAccept!@M, SMDPPReject!@M
States: *idle, CheckStore, storeOK, storeNOK
Attributes: StoreCondition:Boolean; currPID:@M
Traits:
Attribute-Function: idle — {}; CheckStore — {}; storeOK — {}; storeNOK — {};
Transition-Specifications:
    R1: <idle,CheckStore>; SMDPPInvoke(true); true ⟹ currPID′=pid;
    R2: <CheckStore,storeOK>; StoreMsg(true); StoreCondition=true ⟹ true;
    R3: <CheckStore,storeNOK>; StoreMsg(true); StoreCondition=false ⟹ true;
    R4: <storeOK,idle>; SMDPPAccept(currPID=pid); true ⟹ true;
    R5: <storeNOK,idle>; SMDPPReject(currPID=pid); true ⟹ true;
Time-Constraints:
end
```

Figure 81: Formal Specification for MC GRC

7.4.2 The MS class

Figures 82 and 83 show the formal specification for the MS GRC.

7.4.3 The CMS8800 class

Figures 84 and 85 show the formal specification for the CMS8800 GRC.

7.4.4 A subsystem configuration

Figure 86 shows the configuration specification for a subsystem with one MS, one CMS8800 network and one MC.

Class MS [@C]
Events: R-DATA(MSID1.Len,SubID,DestAddr)!@C, R-DATAACCEPT?@C,
  R-DATAREJECT?@C, REORDER?@C, ReleaseAck, ReleaseRej, Timeout2,
  Timeout
States: *DCCHCamping, MOSMSproceeding, MOSMSresend, ProcessAccept,
  ProcessReject, waitresend, wait2
Attributes: Tcnt:Integer; currPID:@C; currMSID:Integer; MSID1:Integer;
  MSID2:Integer
Traits:
Attribute-Function: DCCHCamping — {currPID, Tcnt};
  MOSMSproceeding — {currPID, Tcnt}; MOSMSresend — {};
  ProcessAccept — {}; ProcessReject — {};
  waitresend — {currMSID}; wait2 — {Tcnt};
Transition-Specifications:
  R1: <DCCHCamping,MOSMSproceeding>; R-DATA(true);
    true $\Longrightarrow$ currPID′=pid&CurrMSID′=MSID1&Tcnt′=0;
  R2: <MOSMSproceeding,ProcessAccept>; R-DATAACCEPT(currPID=pid);
    true $\Longrightarrow$ true;
  R3: <MOSMSproceeding,ProcessReject>; R-DATAREJECT(currPID=pid);
    true $\Longrightarrow$ true;
  R4: <MOSMSproceeding,MOSMSresend>; REORDER(currPID=pid);
    MSID2$\neq$0 $\Longrightarrow$ true;
  R5: <MOSMSproceeding,wait2>; R-DATA(true);
    Tcnt=0 $\Longrightarrow$ Tcnt′=1;
  R6: <MOSMSresend,waitresend>; R-DATA(true);
    true $\Longrightarrow$ currMSID′=MSID2;
  R7: <ProcessAccept,DCCHCamping>; ReleaseAck(true);
    true $\Longrightarrow$ currPID′=0;
  R8: <ProcessReject,DCCHCamping>; ReleaseRej(true);
    true $\Longrightarrow$ currPID′=0;
  R9: <waitresend,ProcessReject>; REORDER(currPID=pid);
    true $\Longrightarrow$ true;
  R10: <waitresend,ProcessAccept>; R-DATAACCEPT(currPID=pid);
    true $\Longrightarrow$ true;
  R11: <waitresend,ProcessReject>; R-DATAREJECT(currPID=pid);
    true $\Longrightarrow$ true;
  R12: <waitresend,DCCHCamping>; Timeout2(true);
    true $\Longrightarrow$ true;
  R13: <wait2,DCCHCamping>; Timeout(true);
    true $\Longrightarrow$ Tcnt′=0;

Figure 82: Formal Specification for MS GRC

```
        R14: <wait2.MOSMSresend>: REORDER(currPID=pid);
             MSID2≠0 ⟹ true;
        R15: <wait2.ProcessAccept>: R-DATAACCEPT(currPID=pid);
             true ⟹ true;
        R16: <wait2.ProcessReject>; R-DATAREJECT(currPID=pid);
             true ⟹ true:
Time-Constraints:
        TCvar1: R1. R-DATA. [21. 22]. {MOSMSresend. ProcessAccept. ProcessReject}:
        TCvar3: R6, Timeout2, [21. 22], {ProcessAccept, ProcessReject};
        TCvar2: R5. Timeout. [21. 22]. {MOSMSresend. ProcessAccept. ProcessReject}:
end
```

Figure 83: Formal Specification for MS GRC—continued

```
Class CMS8800 [@S, @MO]
Events: R-DATA(MSID.Len.SubID, DestAddr)?@MO, CheckFeature.
      R-DATAREJECT!@MO, MSIDCheck, SMDPPAccept?@S, SMDPPReject?@S.
      Release, SMDPPInvoke!@S, R-DATAACCEPT!@MO, REORDER!@MO.
      MsgLengthCheck, SubIdentify, DestCheck
States: *idle, OFH, processReject, checkMSID, wait, sendSMS,
      processAccept, processReorder, checkLength,
      IdentifySubscriber, checkDestination
Attributes: MOSMSFeature:Boolean; MAXLENG:Integer; currMO:@MO; currS:@S;
      Subscribers:SubSet; Destinations:DestSet
Traits: Set[Integer,SubSet].Set[Integer,DestSet]
Attribute-Function: idle — {currMO, currS}; OFH — {currMO}; processReject — {};
      checkMSID — {}; wait — {currS}; sendSMS — {}; processAccept — {};
      processReorder — {}; checkLength — {}; IdentifySubscriber — {};
      checkDestination — {};
Transition-Specifications:
      R1: <idle,OFH>; R-DATA(true);
            true ⟹ currMO/=pid;
      R2: <OFH,processReject>; CheckFeature(true);
            MOSMSFeature=false ⟹ true;
      R3: <OFH,checkMSID>; CheckFeature(true);
            MOSMSFeature=true ⟹ true;
      R4: <processReject,idle>; R-DATAREJECT(currMO=pid);
            true ⟹ currMO/=0 & currS/=0;
      R5: <checkMSID,processReorder>; MSIDCheck(true);
            invalidFormat(MSID) ⟹ true;
      R6: <checkMSID,checkLength>; MSIDCheck(true);
            validFormat(MSID) ⟹ true;
      R7: <wait,processAccept>; SMDPPAccept(currS=pid);
            true ⟹ true;
      R8: <wait,processReject>; SMDPPReject(currS=pid);
            true ⟹ true;
      R9: <wait,idle>; Release(true);
            true ⟹ currMO/=0 & currS/=0;
      R10: <sendSMS,wait>; SMDPPInvoke(pid=GetMCport(DestAddr));
            true ⟹ currS/=pid;
      R11: <processAccept,idle>; R-DATAACCEPT(currMO=pid);
            true ⟹ currMO/=0 & currS/=0;
      R12: <processReorder,idle>; REORDER(currMO=pid);
            true ⟹ currMO/=0;
```

Figure 84: Formal Specification for CMS8800 GRC

```
        R13: <checkLength.processReject>: MsgLengthCheck(true);
              Len>MAXLENG ⟹ true;
        R14: <checkLength.IdentifySubscriber>; MsgLengthCheck(true):
              Len≤MAXLENG ⟹ true;
        R15: <IdentifySubscriber,processReject>; SubIdentify(true);
              !(member(SubID,Subscribers)) ⟹ true:
        R16: <IdentifySubscriber,checkDestination>; SubIdentify(true):
              member(SubID.Subscribers) ⟹ true:
        R17: <checkDestination.sendSMS>: DestCheck(true);
              member(DestAddr,Destinations) ⟹ true;
        R18: <checkDestination,processReject>; DestCheck(true);
              member(DestAddr.Destinations) ⟹ true:
Time-Constraints:
        TCvar1: R10. Release. [18. 19], {processReject, processAccept};
end
```

Figure 85: Formal Specification for CMS8800 GRC—continued

```
SCS MOSMS2
        Includes:
        Instantiate:
              MS1::MS[@C:1]:
              Net1::CMS8800[@S:1, @MO:1]:
              MC1::MC[@M:1];
        Configure:
              Net1.@MO1:@MO — MS1.@C1:@C:
              MC1.@M1:@M — Net1.@S1:@S:
end
```

Figure 86: A Subsystem Configuration Specification

7.5 Rose Model without Parameterized Events

Because the current implementation of the TROMLAB tools does not support parameterized events. in order to produce formal specifications compatible to the TROMLAB tools, we flatten the domain of event. that is we replace the R-DATA event by several events, representing all the relevant combinations of parameter values.

We analyzed all the combinations between the relevant values of the necessary parameters (MSID, Bearer data length. Subscriber Identifier, Destination Address), and given the order in which they should be checked by the CMS 8800 network, there are only five distinct combinations:

1. R-DATA with invalid MSID.
2. R-DATA with valid MSID and Length greater than allowed.
3. R-DATA with valid MSID, valid Length and invalid Subscriber Identifier.
4. R-DATA with valid MSID. valid Length, valid Subscriber Identifier and invalid destination address.
5. R-DATA with valid MSID. valid Length, valid Subscriber Identifier and valid destination address.

We will replace the R-DATA event by five events, representing the five combinations listed above.

We also replace the message SMDPP Return Result by two messages:

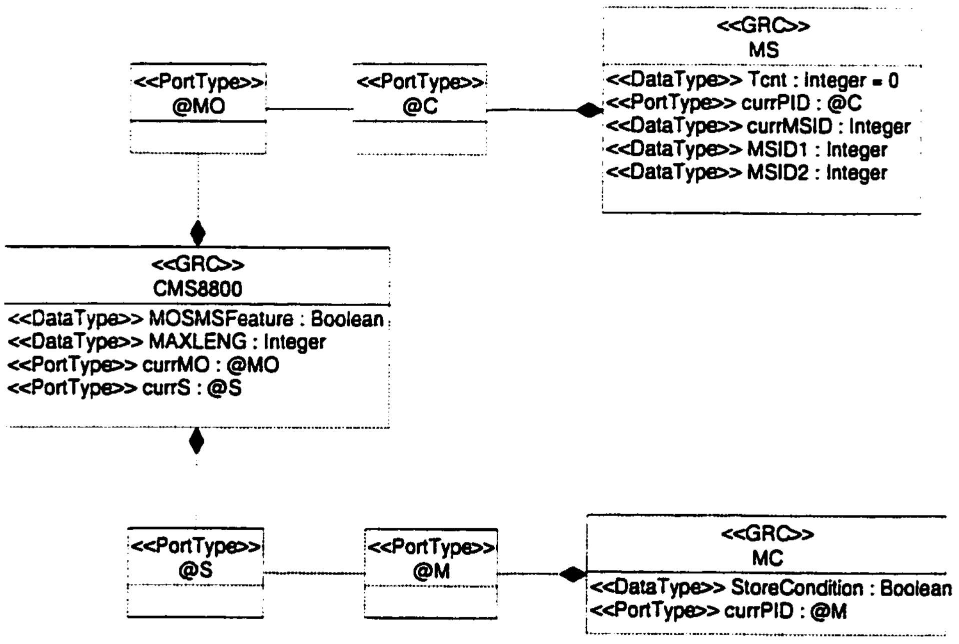
1. SMDPP Accept
2. SMDPP Reject

7.5.1 Class diagram

We model this problem with three generic reactive classes: MS, CMS8800 and MC, and we create a class diagram (Figure 87) for the three GRCs and their respective port types.

Each of the generic reactive classes has the same port types as in the previous model.

However some of the attributes are removed as they are not needed. The sets Subscribers and Destinations are removed from class CMS8800.



<<PortType>>
@MO
<<PortType>>
@C
<<GRC>>
MS
<<DataType>> Tcnt : Integer = 0
<<PortType>> currPID : @C
<<DataType>> currMSID : Integer
<<DataType>> MSID1 : Integer
<<DataType>> MSID2 : Integer
<<GRC>>
CMS8800
<<DataType>> MOSMSFeature : Boolean
<<DataType>> MAXLENG : Integer
<<PortType>> currMO : @MO
<<PortType>> currS : @S
<<PortType>>
@S
<<PortType>>
@M
<<GRC>>
MC
<<DataType>> StoreCondition : Boolean
<<PortType>> currPID : @M

Figure 87: Class diagram with all GRCs for MO SMS

7.5.2 MS GRC

Figure 88 illustrates the statechart diagram for the MS GRC. The behavior of the *MS* class is similar to that in the previous model. The main difference is that instead of sending one R-DATA with four parameters, the *MS* sends five distinct events: R-DATAinvMSID, R-DATAinvLen, R-DATAinvSub, R-DATAinvDest, R-DATAok. When it sends R-DATAinvMSID, it may receive R-DATAREJECT or REORDER. When it sends R-DATAinvLen, R-DATAinvSub or R-DATAinvDest it may receive R-DATAREJECT. When it sends R-DATAok it may receive R-DATAACCEPT, R-DATAREJECT or a timeout may occur, and then the *MS* resends R-DATAok once.

7.5.3 CMS8800 GRC

Figure 89 illustrates the statechart diagram for the CMS8800 GRC. The behavior of the *CMS8800* is similar to that in the previous model. The main difference is that instead of receiving one R-DATA with four parameters, *CMS8800* receives five distinct events: R-DATAinvMSID, R-DATAinvLen, R-DATAinvSub, R-DATAinvDest, R-DATAok. Instead of checking the value of the received parameters against the value of its attributes, *CMS8800* just sends the appropriate event depending on the stimulus received.

7.5.4 MC GRC

Figure 90 illustrates the statechart diagram for the MC GRC. The behavior of the *MC* is identical to that in the previous model.

7.5.5 A subsystem

Figure 91 illustrates the collaboration diagram for a subsystem with one MS, one CMS8800 network and one MC.

7.6 Formal Model without Paramterized Events

7.6.1 The MS class

Figures 92 and 93 show the formal specification for the MS GRC.

7.6.2 The CMS8800 class

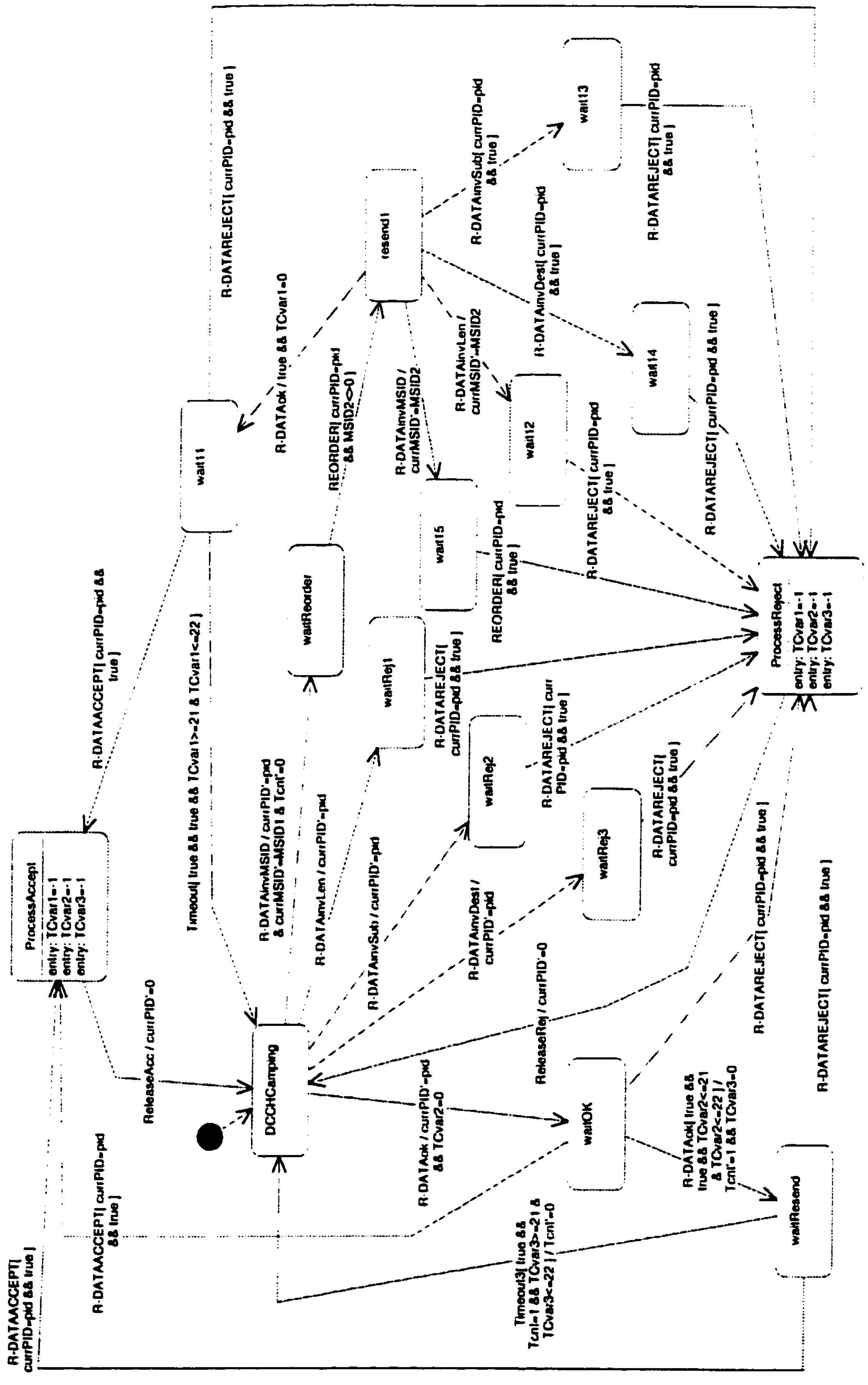
Figures 94 and 95 show the formal specification for the CMS8800 GRC.

7.6.3 The MC class

Figure 96 shows the formal specification for the MC GRC.

7.6.4 A subsystem configuration

Figure 97 shows the configuration specification for a subsystem with one MS, one CMS8800 network and one MC.



DCCHCamping
ProcessAccept
entry: TCvar1=-1
entry: TCvar2=-1
entry: TCvar3=-1
ProcessReject
entry: TCvar1=-1
entry: TCvar2=-1
entry: TCvar3=-1
wait11
wait12
wait13
wait14
wait15
resend1
waitReorder
waitRej1
waitRej2
waitRej3
waitOK
waitResend
ReleaseAcc / currPID=0
ReleaseRej / currPID=0
R-DATAACCEPT[currPID=pid && true]
R-DATAREJECT[currPID=pid && true]
R-DATAAck / true && TCvar1=0
R-DATAAck / currPID=pid && TCvar2=0
REORDER[currPID=pid && MSID2<>0]
REORDER[currPID=pid && true]
R-DATAInvMSID / currMSID'=MSID2
R-DATAInvLen / currMSID'=MSID2
R-DATAInvSub[currPID=pid && true]
R-DATAInvDest[currPID=pid && true]
Timeout[true && true && TCvar1>=21 & TCvar1<=22]
R-DATAInvMSID / currPID'=pid & currMSID'=MSID1 & Tcnt'=0
R-DATAInvLen / currPID'=pid
R-DATAInvSub / currPID'=pid
R-DATAInvDest / currPID'=pid
R-DATAAck[true && true && TCvar2>=21 & TCvar2<=22] / Tcnt'=1 && TCvar3=0
Timeout3[true && Tcnt=1 && TCvar3>=21 & TCvar3<=22] / Tcnt'=0

Figure 88: Statechart diagram for MS GRC

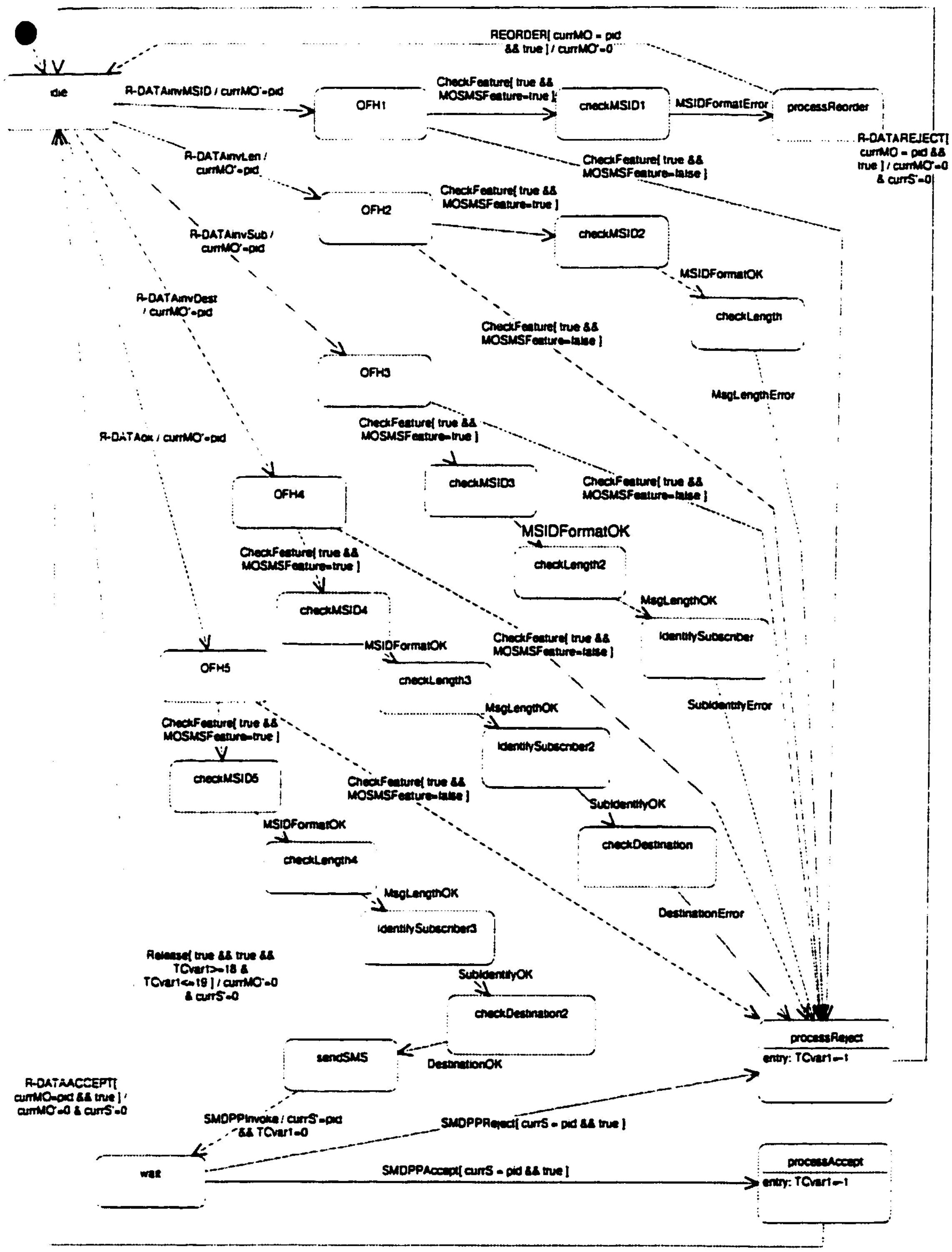

REORDER[currMO = pid && true] / currMO'=0
idle
R-DATAinvMSID / currMO'=pid
OFH1
CheckFeature[true && MOSMSFeature=true]
checkMSID1
MSIDFormatError
processReorder
R-DATAREJECT[currMO = pid && true] / currMO'=0 & currS'=0
R-DATAinvLen / currMO'=pid
CheckFeature[true && MOSMSFeature=false]
OFH2
CheckFeature[true && MOSMSFeature=true]
checkMSID2
R-DATAinvSub / currMO'=pid
MSIDFormatOK
checkLength
R-DATAinvDest / currMO'=pid
CheckFeature[true && MOSMSFeature=false]
OFH3
MsgLengthError
CheckFeature[true && MOSMSFeature=true]
R-DATAok / currMO'=pid
checkMSID3
CheckFeature[true && MOSMSFeature=false]
OFH4
MSIDFormatOK
checkLength2
CheckFeature[true && MOSMSFeature=true]
checkMSID4
MsgLengthOK
IdentifySubscriber
MSIDFormatOK
CheckFeature[true && MOSMSFeature=false]
OFH5
checkLength3
MsgLengthOK
SubIdentityError
CheckFeature[true && MOSMSFeature=true]
IdentifySubscriber2
checkMSID5
CheckFeature[true && MOSMSFeature=false]
SubIdentityOK
MSIDFormatOK
checkDestination
checkLength4
MsgLengthOK
DestinationError
IdentifySubscriber3
Release[true && true && TCvar1>=18 & TCvar1<=19] / currMO'=0 & currS'=0
SubIdentityOK
checkDestination2
processReject
entry: TCvar1=-1
sendSMS
DestinationOK
R-DATAACCEPT[currMO=pid && true] / currMO'=0 & currS'=0
SMDPPInvoke / currS'=pid && TCvar1=0
SMDPPReject[currS = pid && true]
wait
SMDPPAccept[currS = pid && true]
processAccept
entry: TCvar1=-1

Figure 89: Statechart diagram for CMS8800 GRC

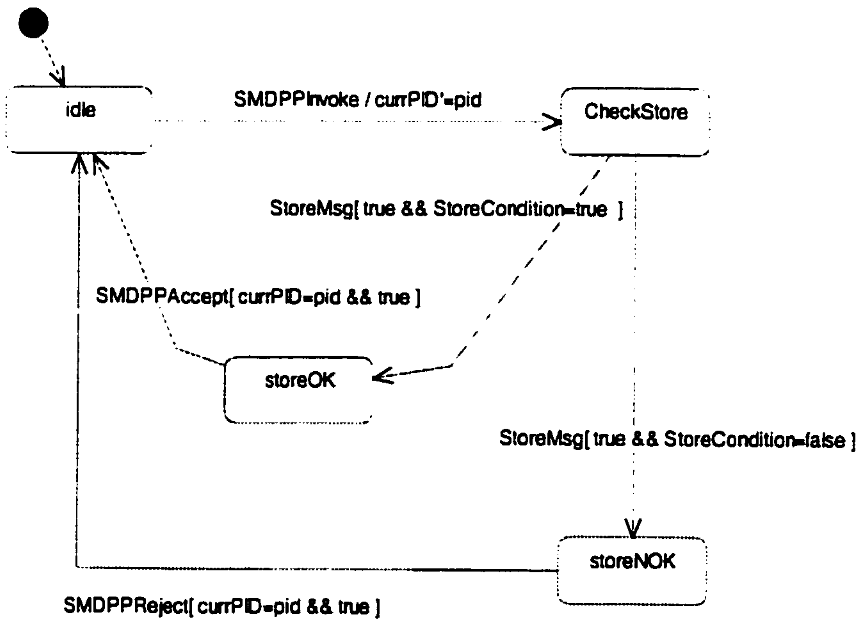

idle
SMDPPInvoke / currPID'=pid
CheckStore
StoreMsg[true && StoreCondition=true]
SMDPPAccept[currPID=pid && true]
storeOK
StoreMsg[true && StoreCondition=false]
storeNOK
SMDPPReject[currPID=pid && true]

Figure 90: Statechart diagram for MC GRC

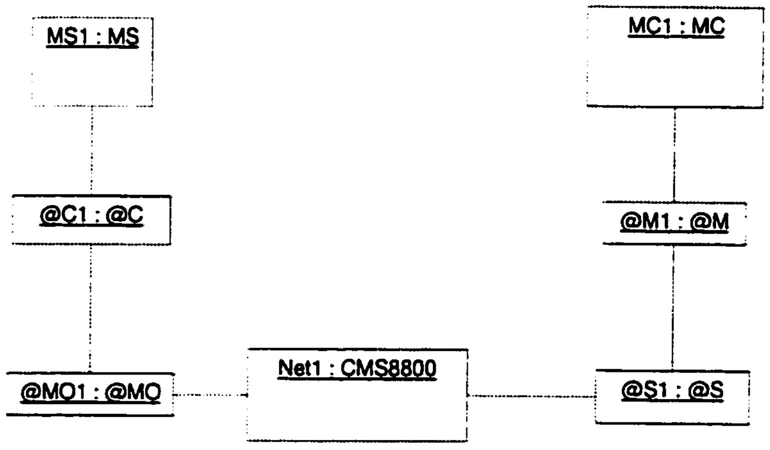


Figure 91: Collaboration diagram for a subsystem

```
Class MS [@C]
Events: R-DATAok!@C, R-DATAinvDest!@C, R-DATAinvSub!@C, R-DATAinvLen!@C,
     R-DATAinvMSID!@C, REORDER?@C, R-DATAREJECT?@C,
     R-DATAACCEPT?@C, Timeout, ReleaseAcc, Timeout3, ReleaseRej
States: *DCCHCamping, waitReorder, waitRej1, waitRej2, waitRej3, waitOK,
     resend1, wait11, ProcessAccept, waitResend, ProcessReject, wait15,
     wait12, wait14, wait13
Attributes: Tcnt:Integer; currPID:@C; currMSID:Integer; MSID1:Integer;
     MSID2:Integer
Traits:
Attribute-Function: DCCHCamping — {currPID, Tcnt};waitReorder — {currPID,
     currMSID, Tcnt};waitRej1 — {currPID};waitRej2 — {currPID};
     waitRej3 — {currPID};waitOK — {currPID};resend1 — {};
     wait11 — {};ProcessAccept — {};waitResend — {Tcnt};
     ProcessReject — {};wait15 — {currMSID};wait12 — {currMSID};
     wait14 — {};wait13 — {};
Transition-Specifications:
R1: <DCCHCamping,waitOK>; R-DATAok(true);
          true ⟹ currPID/=pid;
R2: <DCCHCamping,waitRej3>; R-DATAinvDest(true);
          true ⟹ currPID/=pid;
R3: <DCCHCamping,waitRej2>; R-DATAinvSub(true);
          true ⟹ currPID/=pid;
R4: <DCCHCamping,waitRej1>; R-DATAinvLen(true);
          true ⟹ currPID/=pid;
R5: <DCCHCamping,waitReorder>; R-DATAinvMSID(true);
          true ⟹ currPID/=pid & currMSID/=MSID1 & Tcnt/=0;
R6: <waitReorder,resend1>; REORDER(currPID=pid);
          MSID2≠0 ⟹ true;
R7: <waitRej1,ProcessReject>; R-DATAREJECT(currPID=pid);
          true ⟹ true;
R8: <waitRej2,ProcessReject>; R-DATAREJECT(currPID=pid);
          true ⟹ true;
R9: <waitRej3,ProcessReject>; R-DATAREJECT(currPID=pid);
          true ⟹ true;
R10: <waitOK,ProcessAccept>; R-DATAACCEPT(currPID=pid);
          true ⟹ true;
R11: <waitOK,waitResend>; R-DATAok(true);
          true ⟹ Tcnt/=1;
R12: <waitOK,ProcessReject>; R-DATAREJECT(currPID=pid);
          true ⟹ true;
```

Figure 92: Formal Specification for MS GRC

```
R13: <resend1,wait11>; R-DATAok(true);
         true ⟹ true;
R14: <resend1,wait15>; R-DATAinvMSID(true);
         true ⟹ currMSID/=MSID2;
R15: <resend1,wait12>; R-DATAinvLen(true);
         true ⟹ currMSID/=MSID2;
R16: <resend1,wait14>; R-DATAinvDest(currPID=pid);
         true ⟹ true;
R17: <resend1,wait13>; R-DATAinvSub(currPID=pid);
         true ⟹ true;
R18: <wait11,DCCHCamping>; Timeout(true);
         true ⟹ true;
R19: <wait11,ProcessAccept>; R-DATAACCEPT(currPID=pid);
         true ⟹ true;
R20: <wait11,ProcessReject>; R-DATAREJECT(currPID=pid);
         true ⟹ true;
R21: <ProcessAccept,DCCHCamping>; ReleaseAcc(true);
         true ⟹ currPID/=0;
R22: <waitResend,DCCHCamping>; Timeout3(true);
         Tcnt=1 ⟹ Tcnt/=0;
R23: <waitResend,ProcessAccept>; R-DATAACCEPT(currPID=pid);
         true ⟹ true;
R24: <waitResend,ProcessReject>; R-DATAREJECT(currPID=pid);
         true ⟹ true;
R25: <ProcessReject,DCCHCamping>; ReleaseRej(true);
         true ⟹ currPID/=0;
R26: <wait15,ProcessReject>; REORDER(currPID=pid);
         true ⟹ true;
R27: <wait12,ProcessReject>; R-DATAREJECT(currPID=pid);
         true ⟹ true;
R28: <wait14,ProcessReject>; R-DATAREJECT(currPID=pid);
         true ⟹ true;
R29: <wait13,ProcessReject>; R-DATAREJECT(currPID=pid);
         true ⟹ true;
Time-Constraints:
TCvar2: R1. R-DATAok. [0. 22]. {ProcessAccept, ProcessReject};
TCvar1: R13. Timeout. [21. 22]. {ProcessAccept, ProcessReject};
TCvar3: R11. Timeout3. [21. 22]. {ProcessAccept. ProcessReject};
end
```

Figure 93: Formal Specification for MS GRC—continued

Class CMS8800 [@S, @MO]
Events: R-DATAinvMSID?@MO, R-DATAinvLen?@MO, R-DATAinvSub?@MO,
R-DATAinvDest?@MO, R-DATAok?@MO, CheckFeature, R-DATAREJECT!@MO,
SMDPPAccept?@S, SMDPPReject?@S, Release, SMDPPInvoke!@S,
R-DATAACCEPT!@MO, REORDER!@MO, MsgLengthError, SubIdentifyError,
DestinationError, MsgLengthOK, MSIDFormatOK, MSIDFormatError,
SubIdentifyOK, DestinationOK
States: *idle, OFH1, processReject, wait, sendSMS, processAccept,
processReorder, checkLength, IdentifySubscriber, checkDestination,
OFH2, checkLength2, OFH3, checkMSID3, checkMSID2, checkMSID1,
IdentifySubscriber2, checkLength3, checkMSID4, OFH4, OFH5,
checkMSID5, checkLength4, IdentifySubscriber3, checkDestination2
Attributes: MOSMSFeature:Boolean; MAXLENG:Integer; currMO:@MO: currS:@S
Traits:
Attribute-Function: idle — {currMO, currS}; OFH1 — {currMO}; processReject — {};
wait — {currS}; sendSMS — {}; processAccept — {};
processReorder — {}; checkLength — {}; IdentifySubscriber — {};
checkDestination — {}; OFH2 — {currMO}; checkLength2 — {};
OFH3 — {currMO}; checkMSID3 — {}; checkMSID2 — {};
checkMSID1 — {}; IdentifySubscriber2 — {}; checkLength3 — {};
checkMSID4 — {};OFH4 — {currMO};OFH5 — {currMO};
checkMSID5 — {};checkLength4 — {};IdentifySubscriber3 — {};
checkDestination2 — {};
Transition-Specifications:
R1: <idle,OFH1>; R-DATAinvMSID(true); true $\Longrightarrow$ currMO'=pid;
R2: <idle,OFH2>; R-DATAinvLen(true); true $\Longrightarrow$ currMO'=pid;
R3: <idle,OFH3>; R-DATAinvSub(true); true $\Longrightarrow$ currMO'=pid;
R4: <idle,OFH4>; R-DATAinvDest(true); true $\Longrightarrow$ currMO'=pid;
R5: <idle,OFH5>; R-DATAok(true); true $\Longrightarrow$ currMO'=pid;
R6: <OFH1,processReject>; CheckFeature(true); MOSMSFeature=false $\Longrightarrow$ true;
R7: <OFH1,checkMSID1>; CheckFeature(true); MOSMSFeature=true $\Longrightarrow$ true;
R8: <processReject,idle>; R-DATAREJECT(currMO=pid);
true $\Longrightarrow$ currMO'=0 & currS'=0;
R9: <wait,processAccept>; SMDPPAccept(currS=pid); true $\Longrightarrow$ true;
R10: <wait,processReject>; SMDPPReject(currS=pid); true $\Longrightarrow$ true;
R11: <wait,idle>; Release(true); true $\Longrightarrow$ currMO'=0 & currS'=0;
R12: <sendSMS,wait>; SMDPPInvoke(true); true $\Longrightarrow$ currS'=pid;
R13: <processAccept,idle>; R-DATAACCEPT(currMO=pid);
true $\Longrightarrow$ currMO'=0 & currS'=0;
R14: <processReorder,idle>; REORDER(currMO=pid); true $\Longrightarrow$ currMO'=0;
R15: <checkLength,processReject>; MsgLengthError(true); true $\Longrightarrow$ true;

Figure 94: Formal Specification for CMS8800 GRC

```
     R16: <IdentifySubscriber,processReject>: SubIdentifyError(true); true ⟹ true;
     R17: <checkDestination,processReject>: DestinationError(true); true ⟹ true;
     R18: <OFH2,processReject>; CheckFeature(true); MOSMSFeature=false ⟹ true;
     R19: <OFH2,checkMSID2>; CheckFeature(true); MOSMSFeature=true ⟹ true;
     R20: <checkLength2,IdentifySubscriber>; MsgLengthOK(true); true ⟹ true;
     R21: <OFH3,processReject>: CheckFeature(true): MOSMSFeature=false ⟹ true;
     R22: <OFH3,checkMSID3>; CheckFeature(true): MOSMSFeature=true ⟹ true:
     R23: <checkMSID3.checkLength2>; MSIDFormatOK(true); true ⟹ true;
     R24: <checkMSID2,checkLength>; MSIDFormatOK(true); true ⟹ true;
     R25: <checkMSID1,processReorder>: MSIDFormatError(true); true ⟹ true;
     R26: <IdentifySubscriber2.checkDestination>; SubIdentifyOK(true); true ⟹ true;
     R27: <checkLength3.IdentifySubscriber2>; MsgLengthOK(true): true ⟹ true:
     R28: <checkMSID4.checkLength3>; MSIDFormatOK(true); true ⟹ true:
     R29: <OFH4.checkMSID4>: CheckFeature(true): MOSMSFeature=true ⟹ true:
     R30: <OFH4.processReject>: CheckFeature(true): MOSMSFeature=false ⟹ true:
     R31: <OFH5.checkMSID5>; CheckFeature(true); MOSMSFeature=true ⟹ true:
     R32: <OFH5,processReject>: CheckFeature(true); MOSMSFeature=false ⟹ true:
     R33: <checkMSID5.checkLength4>; MSIDFormatOK(true); true ⟹ true;
     R34: <checkLength4.IdentifySubscriber3>: MsgLengthOK(true); true ⟹ true;
     R35: <IdentifySubscriber3.checkDestination2>: SubIdentifyOK(true): true ⟹ true;
     R36: <checkDestination2.sendSMS>: DestinationOK(true): true ⟹ true:
Time-Constraints:
     TCvar1: R12. Release. [18. 19], {processReject. processAccept}:
end
```

Figure 95: Formal Specification for CMS8800 GRC—continued

```
Class MC [@M]
Events: SMDPPInvoke?@M, StoreMsg, SMDPPAccept!@M, SMDPPReject!@M
States: *idle, CheckStore, storeOK, storeNOK
Attributes: StoreCondition:Boolean;currPID:@M
Traits:
Attribute-Function: idle → {};CheckStore → {currPID};storeOK → {};storeNOK → {};
Transition-Specifications:
    R1: <idle,CheckStore>; SMDPPInvoke(true); true ⟹ currPID′=pid;
    R2: <CheckStore,storeOK>; StoreMsg(true); StoreCondition=true ⟹ true;
    R3: <CheckStore,storeNOK>; StoreMsg(true); StoreCondition=false ⟹ true;
    R4: <storeOK,idle>; SMDPPAccept(currPID=pid); true ⟹ true;
    R5: <storeNOK,idle>; SMDPPReject(currPID=pid); true ⟹ true;
Time-Constraints:
end
```

Figure 96: Formal Specification for MC GRC

```
SCS MS-CMS8800-MC-1
    Includes:
    Instantiate:
        MS1::MS[@C:1];
        Net1::CMS8800[@S:1, @MO:1];
        MC1::MC[@M:1];
    Configure:
        Net1.@MO1:@MO — MS1.@C1:@C;
        MC1.@M1:@M — Net1.@S1:@S;
end
```

Figure 97: A Subsystem Configuration Specification

7.7 Conclusion

The first model for mobile originating short message services shows that this type of application may be modeled using this technique. The translator is capable to use parameterized events. but the formal specifications created are based on an example [AAR95] and not on a well-defined grammar.

The second model for mobile originating short message services shows that in order to use the formal specifications in the TROMLAB environment. it is possible to replace a parameterized event with a set of events representing the relevant combinations between values of all the parameters. Even if the necessary number of paramters was low and a lot of duplicate combinations were eliminated, it proved to be a tedious process. The number of states. events and transition specifications in the resulting state machines has exploded, making the model hard to understand and error-prone.

We may conclude that adding parameterized events in the TROMLAB environment would make the technique and tools capable of modeling a small industrial application. such as MO SMS.

Chapter 8

Future Enhancements

8.1 Introduction

This chapter presents a few suggested enhancements to TROMLAB and the Rose-GRC Translator. We discuss the integration of Rose and the translator with TROMLAB. We also explain the link between the Rose-GRC Translator and a future verifier tool in TROMLAB, and we discuss the usage of UML sequence diagrams for generic reactive systems. We propose a few enhancements to the translator. Last, but not least we discuss the need for the introduction of parameterized events in the model and comment on how the translator can be adapted for this.

8.2 Integration of the Rose-GRC Translator in TROMLAB

In order to provide a seamless interface between the Rose-GRC Translator and TROMLAB, we must address the issue of integration. The version of Rose that was available for development was for a Windows NT environment. TROMLAB is available on Unix. A single development platform is desirable to ensure a consistent interface between the two.

If TROMLAB could be executed on Windows NT, then it would be possible to trigger the parser, semantic analyzer, simulator and a future verifier from Rose. Rose's menus may be customized by modifying the file Rose.mnu, in order to add commands that execute these programs. See section 5.4.3 for how to customize the Rose menu.

Alternatively, if Rose for Unix becomes available, Rose could be started by the graphic user interface of TROMLAB that is currently being developed.

At the present time, because of the platform difference, we propose the following. The user develops the Rose model on Windows NT, which creates one or several text files. When the translator asks the user for the file names and path, the user may select a Unix directory, if it is available on the network. If it is not available on the network, the user would have to copy the files from the PC to a diskette and from here to a Unix directory, and finally transferred to Unix files. The TROMLAB tools may read the files thus obtained.

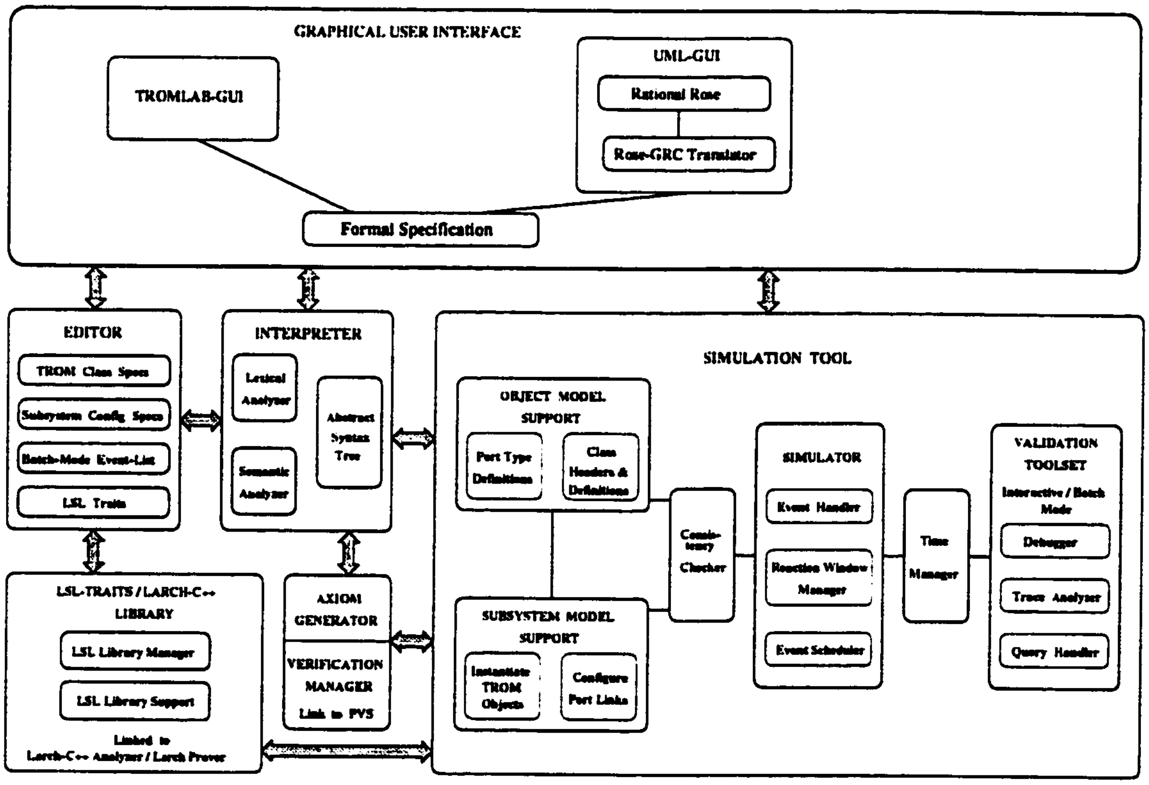


Figure 98: Integration of the UML-GUI in the TROMLAB architecture

By using Rose for visual modeling and the Rose-GRC Translator to obtain the underlying formal model, we provide an alternative to the graphical user interface of TROMLAB. Even if Rose and TROMLAB are on different platforms, we may abstract the graphical user interface component of the TROMLAB architecture to two subcomponents: the UML-GUI and the TROMLAB-GUI [Sri99], as shown in Figure 98.

For the time being, UML-GUI and TROMLAB-GUI are mutually exclusive, but a link may be developed in the future, if required. Both UML-GUI and TROMLAB-GUI create formal specifications which are the input to TROMLAB. The UML-GUI comprises the Rose tool and the Rose-GRC Translator.

8.3 A Link to Verification

This section explains the link between the work presented in this thesis and ongoing work on integrating UML and PVS(Prototype Verification System)[ORS92] for automated reasoning. The work [AM98], [Pom99] involves embedding the formal specifications of a generic reactive system in PVS, and using PVS for proving system properties.

One of the aspects of verification is to ensure the correctness of a scenario. The sequence of messages in a scenario must be verified against the state machines of the objects involved. If the time of occurrence is specified for each message, then this must be verified against the timing constraints given on the state transitions of the objects involved.

In order to verify the correctness of a particular scenario, an ordered list of message entries has to be given to a future verifier tool. A message entry is composed of the sending object, the message name, the receiving object, and, optionally, the time of occurrence.

A particular scenario may be visually modeled in UML in a sequence diagram. A UML sequence diagram is a type of interaction diagram that traces the execution of a scenario in time. It contains objects, their time lines and messages. The sequence of the messages may or may not be numbered.

The following information may be extracted from a Rose sequence diagram: sending object, message name, receiving object.

Ideally, for real-time reactive systems, the time of occurrence of a message should be shown on the sequence diagram, near the message itself. In Rose this could be done by adding the time as a string and attaching this string to the message, through Rose's option "Attach script" from the Edit menu. Figure 99 shows such an ideal sequence diagram. However, the translator cannot extract this string from the Rose model. For this reason, we propose to not show the time of occurrence on the sequence diagram (see Figure 100), but rather to allow the user the possibility to enter it during translation. See section 5.4.3 for details on how the user may enter the time of occurrence.

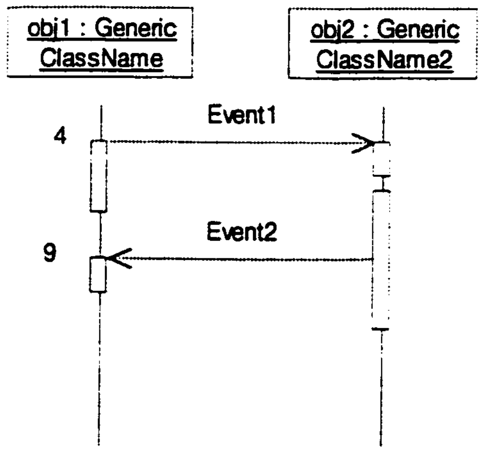


Figure 99: A short sequence diagram with time of occurence as text attached to the message

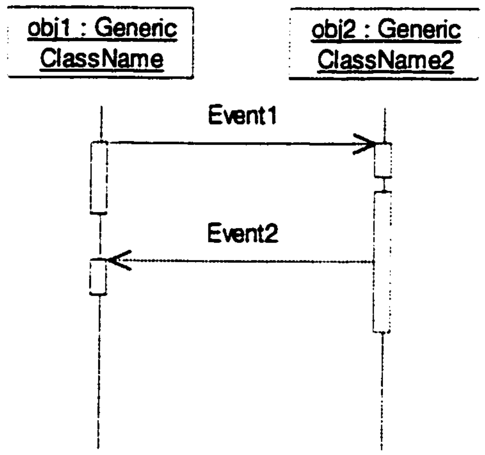


Figure 100: A short sequence diagram without time of occurence

8.4 Enhancements to Translator

Rose-GRC translator in Java

The Rose-GRC Translator may be implemented in Java, for a smoother integration with the other TROMLAB tools. This was considered when implementing this version of the translator, however not much information was available on the interface between the Rose classes and Java. Implementing it in Java would have the benefit of using dynamic structures, which are limited in RoseScript.

Rose on Unix
When Rose is available for Unix, the Rose-GRC Translator should be ported to the Unix platform. This may need some modifications mainly in the area of the graphical user interface. Having Rose and the translator on Unix will facilitate the integration of Rose-GRC Translator and the Interpreter in TROMLAB.

8.5 Parameterized Events

The need to support parameterized events in the formalism became obvious after modeling the MO SMS example. Parameters carry data between objects.

Rose supports paramterized events, thus the model of a generic reactive system can be easily modified. Arguments may be entered by the user through Rose's state transition dialog, and may be used in all the assertions, whenever the event is used.

The Rose-GRC Translator was designed in such a way as to allow an easy addition of parameters, as follows:

- Has an option selection box that allows the user to select if parameterized events are supported.
- Has a boolean flag, ParameterFlag, which may be used in order to extract the arguments of events from Rose and store them in its internal structure.
- Has provision in its internal structure to store arguments of each event.
- The translator does not parse the assertions on the transition specifications and it only forwards them to the TROMLAB interpreter, therefore the usage of parameters in these assertions is transparent.
- The subroutines PrintEvents and PrintTransitionSpecs can be easily modified to print also the parameters from the translator's internal structure, once the grammar is updated with parameterized events.

Chapter 9

Conclusion

By improving the UML model introduced in [AM98] for generic reactive systems, and integrating it with the formal model, we have provided a development technique that has the advantages of both visual modeling and of formal modeling.

The modeling technique, although mainly illustrated for real-time reactive systems, may be **adapted** to any generic reactive system, without specific time constraints but only with temporal ordering dictated by the statechart diagrams.

We developed the Rose-GRC Translator as a **proof of concept**, using RoseScript, the language provided by the Rose Extensibility Interface (REI). Due to some language restrictions imposed by RoseScript in the area of user-defined structures, there is a size limitation on the number of components in a generic reactive class, but there is no restriction as to the number of classes in the model. **Scalability** is ensured by the algorithms, only data structures need to be improved.

Application specialists are usually reluctant to use rigorous formal notations at a modeling level due to the rather heavy and constrained constructs. We provide the **flexibility** of modeling with a widely accepted industrial standard tool, Rose.

Application specialists may model a generic reactive system first from a business perspective in UML using use cases, and then from a software system perspective, using class diagrams, statechart diagrams, collaboration and sequence diagrams. If the software system is modeled as described in this thesis, then the Rose-GRC Translator may be used to obtain the corresponding formal model of the system. The formal model may be used by the TROMLAB family of tools to **validate** the model using the semantic analyzer [Tao96] and the simulator [Mut96], and to prove certain critical properties using a future verifier, based on PVS, the formal reasoning system. Simulating the behavior of the generic model with the existing TROMLAB simulator [Mut96] ensures that the model is validated at the early phases of design, prior to committing to an implementation.

The Rose-GRC Translator provides an intuitive standard **graphical user** interface similar to Rose's. The developer has **support** from Rose's Help facilities. The Rose-GRC Translator provides numerous syntactic and consistency checks on the model and gives a set of comprehensive error messages through its graphical interface.

By developing the Rose-GRC Translator we demonstrate that even if the developer chooses to use a visual modeling tool such as Rose, with a few modeling guidelines, the underlying formal model may be obtained automatically to provide the base for rigorous simulation and formal verification.

Bibliography

[AAM96] V. S. Alagar, R. Achuthan, and D. Muthiayen. TROMLAB: A software development environment for real-time reactive systems. Submitted for publication in *ACM Transactions on Software Engineering and Methodology (Being revised)*, October 1996.

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

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

[AM98] V. S. Alagar and D. Muthiayen. Specification and verification of complex real-time reactive systems modeled in UML. Submitted for publication in *IEEE Transactions on Software Engineering (Being revised)*. July 1998.

[Eri98] Ericsson Research Canada. *Network Function Specification: CMS 8800/ Mobile Originating Short Message Services on DCCH*. March 1998.

[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, March 1999. Under preparation.

[HL94] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proceedings of the*

15th IEEE Real-Time Systems Symposium, RTSS'94, pages 120-131, San Juan, Puerto Rico, December 1994.

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

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

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

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

[Obj97] Overcoming the crisis in real-time software development. Technical report, ObjecTime Limited, 1997.

[Obj98] Uml for real-time overview. Technical report, ObjecTime Limited, 1998.

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

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

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

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

[Rat98b] Rational Software Corporation. *Rational Rose 98 Using Rose*, February 1998.

[RS98] J. Rumbaugh and B. Selic. Using uml for modeling complex real-time systems. Technical report, March 1998.

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

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

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

[TIA97] TIA/EIA. *TIA/EIA-41.5-D Cellular Radiotelecommunications Intersystem Operations*, December 1997.

[TIA98] TIA/EIA. *TIA/EIA-136-123, Digital Control Channel Layer 3*, October 1998.

Appendix A

Data Structures in the Rose-GRC Translator

RoseScript supports user-defined types. We declared several types for the translator's internal structures for GRC specifications, for SCS specifications and for message sequencing.

For most elements of a Rose model there is a corresponding collection. Rose Extensibility provides a set of properties and methods that allows access to a particular element in any collection.

In RoseScript, collections cannot be defined for user-defined types, therefore arrays had to be used to hold elements of the same type. Arrays can be fixed or dynamic. Within user-defined structures, only fixed arrays are permitted. This imposes a restriction on the number of components of the same type that a GRC may have, but does not restrict the number of generic classes in the Rose model.

We declared the following **types**:

```
Type GRCPortType
   PortTypeClass As Class     'Rose class representing the PortType
   EventSet As String         'stores the list of events extracted from the
                              'attribute events of port type class
End Type


Type GRCEvent
   evName As String           ' event name
   evType As String           '""=internal; "?"=input; "!"=output
   evPortType As Integer      'index in the PortTypes array, indicating
                              ' at which port type the event may happen
   evArguments As String      ' Future: list of arguments for the event
End Type
```

```
Type GRCAttribute
    Name As String              ' attribute name
    Type As String              ' attribute type. Can be simple type Integer
                                ' or Boolean or can be extracted from an LSL trait
    InitVal As String           ' Initial value of the attribute. Usually empty
    Stereotype As String        ' stereotype of the attribute: PortType or DataType
End Type


Type GRCState
    State As State              ' State name
    isInitial As Boolean        ' if true, this state is the initial state
                                ' for the context
    isComplex As Boolean        ' if true, this state has substates
    NumAttributesInFunction As Integer
                                'number of attributes in attribute function
    AttribFunc(MaxAttributes) As Integer
                                'array of indexes to the GRCAttributes array,
                                'indicating which attributes are in the attribute
                                'function of this state
End Type


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.
End Type


Type GRCTimeConstraint
    TCVar As String             ' Name of the time constraint, same as the TC
                                ' variable in the Rose model
    ConstrainingTransPos As Integer
                                ' Position in the GRCTransitions array of
                                ' the constraining transition for this time constraint
    ConstrainedTransPos As Integer
                                ' Position in the GRCTransitions array of the
                                ' constrained transition for this time constraint
    TCLowerBound As String
                                ' Lower bound for the time constraint interval,
                                ' represented as string for convenience
    TCLowerBoundType As String
                                ' String representing open or closed interval
                                ' for lower bound
    TCUpperBoundType As StringString representing open or closed interval
                                ' for upper bound
    TCUpperBound As String      ' Upper bound for the time constraint interval,
                                ' represented as string for convenience
    DisablingStatePos(MaxStates) As Integer
                                ' position in the GRCStates array
                                ' for all states which are disabling states
                                ' for this transition
    NumDisablingStates As Integer
                                ' Number of valid entries in array
                                ' DisablingStatesPos()
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
    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


Type GRCObject
    Name As String                ' name of Rose object
    GRCPosition As Integer        ' position in GRCArray of class corresponding to object
    GRCPortCardinality(MaxPortTypes) As Integer
                                  ' at position i: number of ports of port type at
                                  ' position i in array PortTypes for the
                                  ' GRC corresponding to the object
    End Type



Type GRCPortLink
    Obj1Name As String            ' object name of object linked to port 1 of the link
    Obj2Name As String            ' object name of object linked to port 1 of the link
    Port1Name As String           ' name of port 1 on the link
    PortType1Name As String       ' name of port type of port 1 on the link
    Port2Name As String           ' name of port 2 on the link
    PortType2Name As String       ' name of port type of port 2 on the link
End Type
```

Appendix B

Rose Classes with Properties and Methods

We list here the properties and methods of the Rose classes and methods used by the Rose-GRC Translator, taken from [Rat98a].

Element Class	
Property	Description
Name	Name of a model element
Method	Description

RoseItem Class	
Property	Description
Stereotype	Specifies the Roseitem's stereotype
Method	Description

Class Class	
Property	Description
Attributes	Collection that contains the attributes belonging to the class
StateMachine	State machine that belongs to the class
Method	Description
GetRoles	Retrieves the roles belonging to the class

Role Class	
Property	Description
Association	Specifies an association belonging to the role
Aggregate	Indicates whether the role is an aggregate class
Method	Description

Association Class	
Property	Description
Method	Description
GetOtherRole	Retrieves another role from an association

Attribute Class	
Property	Description
InitValue	Initial value of the attribute
Type	Type of the attribute
Method	Description

Transition Class	
Property	Description
Method	Description
GetSourceState	Retrieves the transition's initial state
GetTargetState	Retrieves the transition's target state
GetTriggerAction	Retrieves the action to perform when the transition's trigger event occurs
GetTriggerEvent	Retrieves the event that triggers the transition

Event Class	
Property	Description
Arguments	Conditions affecting the event
GuardCondition	Defines a condition which, when true, causes the event to occur. As long as the condition remains false, the event will not occur.
Name	Name of the event
Method	Description

StateMachine Class	
Property	Description
Method	Description
GetAllStates	Retrieves all states belonging to a state machine
GetAllTransitions	Retrieves all transitions belonging to a state machine
GetTransitions	Retrieves the collection of transitions belonging to the state machine

State Class	
Property	Description
ParentState	State (superstate) that contains the state
StateKind	Rich type that indicates the type of state
Substates	Specifies the substates of the stateNote: A state with substates is called a superstate.
Transitions	Specifies the set of transitions defined for the state
Method	Description
GetEntryActions	Retrieves the entry actions belonging to a state

Action Class	
Property	Description
Method	Description

ClassDiagram Class	
Property	Description
Method	Description
GetClasses	Retrieves the collection that contains the classes belonging to the class diagram
IsUseCaseDiagram	Determines whether the class diagram is a use case diagram

ScenarioDiagram Class	
Property	Description
Method	Description
GetDiagramType	Retrieves the value of the type of diagram
GetMessages	Retrieves the collection of messages that belong to the scenario diagram

Link Class	
Property	Description
LinkRole1	Defines an object instance as a Role1 link
LinkRole2	Defines an object instance as a Role2 link
Method	Description

ObjectInstance Class	
Property	Description
Method	Description
GetClass	Retrieves the class to which the object instance belongs

Message Class	
Property	Description
Method	Description
GetSenderObject	Retrieves the object that sent the message
GetReceiverObject	Retrieves the object that received the message

Collections Class	
Property	Description
Count	Number of objects in a collection
Method	Description
GetAt	Retrieves a specified instance of an object in a given collection