SIMULATED VALIDATION OF REAL-TIME REACTIVE SYSTEM WITH PARAMETERIZED EVENTS

SHI HUI LIU

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

Your file Votre référence
ISBN: 0-612-83912-5
Our file Notre référence
ISBN: 0-612-83912-5

Abstract

Simulated Validation of Real-Time Reactive System with Parameterized Events

Shi Hui Liu

This thesis addresses the problems encountered during the simulated validation of real-time reactive systems at the design phase before the implementation. We assume that such systems are modeled as timed labeled transition systems, following the TROM methodology with parameterized events. In the previous work on the simulation, the simulator's algorithm had problems simulating some real-time reactive systems, such as the Train-Gate-Controller and Robotics system. This thesis aims to correct the flaws related to time conflicts in previous models, separate the Data Model module from the Validation tool, and add support for parameterized events in the simulator. As a system grows larger, assessing the performance of mission-critical applications become more important. This thesis proposes an assessment of a system's performance in terms of functionality, based on a Simulation, and illustrates its approach using the Robotics case study.

To my family

Acknowledgments

I would like to thank my supervisors, Dr. V.S. Alagar and Dr. O. Ormandjieva, for their technical and financial support throughout my studies. They have guided my work with good advice and insightful comments.

I would also like to thank my family for encouraging my pursuit of higher education. They have always provided me with comfort and support whenever I encounter any problems.

Contents

List of Figures .... **ix**

List of Tables .... **xii**

1 Introduction **1**
1.1 Real-Time Reactive Systems 1
1.2 Simulated Validation 3
1.3 Related Work 3
1.4 Major Contributions 3
1.5 Thesis Outline 4
2 TROMLAB Environment – a Brief Review of Initial Design **5**
2.1 TROM Formalism 5
2.1.1 Data Abstraction Tier 6
2.1.2 TROM Tier 7
2.1.3 Operational Semantics 10
2.1.4 Subsystem Tier 12
2.2 SimulationValidation Components in TROMLAB 12
2.2.1 The Interpreter 12
2.2.3 The Simulation Tool 13
3 Time Interval Comparison Algorithm **16**
3.1 Motivation 16
3.2 Train-Gate-Controller 16
3.3 Sample Simulation Event List 16
3.4 Simulation Result of the Present Simulator 17
3.4 Problem Analysis 19
3.5 Time Interval Comparison Algorithm 19
3.5.1 Overlaps 21
3.5.2 Meet 21
3.5.3 Before 21
3.5.4 During 21
3.5.5 Start 21
3.5.6 Finish 22
3.5.7 Equal 22
3.5.8 Simulation Result with Modified Algorithm 22
4 Data Model Separation **24**
4.1 Motivation 24
4.2 Previous Design 24
4.3 A Design Pattern – Façade 25
4.4 New Design 26
5 Parameterized Events **27**
5.1 Review the Parameterized Events 27
5.1.1 Syntax of Parameterized Events 28
5.1.2 Semantics of Parameterized Events 30
5.2 Extensions to the Parameterized Events 31

5.2.1 Extension to the Type of Parameter 31
5.2.2 Extension to the Syntax of Simulation Event List 32
5.3 Upgraded Interpreter 33
5.3.1 The Previous Design with Parameterized Events 33
5.3.2 Upgraded Interpreter 35
5.4 Upgraded Simulator 38
5.4.1 Previous Design 38
5.4.2 New Design 41
5.4.3 Simulation Algorithm 44
6 Train-Gate-Controller Case Study 48
6.1 Problem Description 48
6.2 Original TGC System 49
6.2.1 Class Diagram for TGC System 49
6.2.2 Train Class 49
6.2.3 Controller Class 50
6.2.4 Gate Class 52
6.2.5 System Configuration Specification (SCS) 53
6.2.6 LSL trait 54
6.3 Remodeling the TGC System with Parameterized Events 54
6.3.1 Class Diagram for TGC System with Parameterized Events 55
6.3.2 Train Class with Parameterized Events 55
6.3.3 Controller Class with Parameterized Events 56
6.3.4 Gate Class with Parameterized Events 57
6.3.5 Simulation Event List with Parameterized Events 58
6.3.6 Simulation Result with Parameterized Events 58
7 Robotic Assembly Case Study 60
7.1 Problem Description 60
7.2 Assumptions and Time Constraints 60
7.3 Original Modeling 61
7.3.1 Class Diagram for Robotics System 61
7.3.2 Formal Problem Description 63
7.3.3 Simulation Result Analysis 72
7.3.4 Explanation of the Result 72
7.3.5 Existing Problems in the Design 72
7.3.6 Summary of problems 75
7.4 Remodeling 75
7.4.1 Robotics System with a Self-controlled User 76
7.4.2 Robotics System with Buffered Belt 82
7.5 Remodeling with Parameterized Events 91
7.5.1 The LSL Trait – Part 91
7.5.2 Self-controlled User Modeling with Parameterized Events 91
7.5.3 Buffered Belt Modeling with Parameterized Events 99
8 Simulation-Based Measurement of System Functionality 107
8.1 Functionality Measure 107
8.2 Formulas 108
8.3 Illustration on the Robotics Case Study 109
9 Conclusions and Future Work 111
9.1 Work Synthesis 111
9.2 Future Work 112
9.2.1 Parameterized Events 112

9.2.2 Simulator 112
Bibliography 115
Appendix A 117
Appendix B 119
Appendix C 122

List of Figures

Figure 1: Architecture of TROMLAB *2*
Figure 2: Three Tiers *6*
Figure 3: Set trait *6*
Figure 4: TROM class anatomy *10*
Figure 5: Architecture of interpreter *13*
Figure 6: Architecture of Validation tool *15*
Figure 7: Sample Simulation Event List *17*
Figure 8: The simulating result of present Simulator *18*
Figure 9: Allen's temporal predicates *20*
Figure 10: Simulating result with modified simulation algorithm *23*
Figure 11: The relation between Data-related classes in previous design *25*
Figure 12: Façade pattern *25*
Figure 13: New design with Façade pattern *26*
Figure 14: New TROM class anatomy *30*
Figure 15: Interpreter Class Diagram (old) *33*
Figure 16: Interpreter Class Diagram – Detailed (old) *34*
Figure 17: Interpreter Class Diagram – TROMclass (old) *34*
Figure 18: Interpreter Class Diagram – SCS (old) *35*
Figure 19: Interpreter Class Diagram (new) *37*
Figure 20: Interpreter Class Diagram – Detailed (new) *37*
Figure 21: Interpreter Class Diagram – TROMclass (new) *38*
Figure 22: Simulator Class Diagram (old) *39*
Figure 23: Simulator Class Diagram – Subsystem (old) *39*
Figure 24: Simulator Class Diagram – TROM (old) *40*
Figure 25: Simulator Class Diagram – SimulationEvent (old) *41*
Figure 26: Simulator Class Diagram – SimulationEvent (new) *42*
Figure 27: Simulator Class Diagram – SimulationEvent (new) *43*
Figure 28: TGC system class diagram *49*
Figure 29: Train TROM class – Class specification *50*
Figure 30: Train TROM class – State chart diagram *50*
Figure 31: Train TROM class – class diagram *50*
Figure 32: Controller TROM class – Class specification *51*
Figure 33: Controller TROM class – State chart diagram *51*
Figure 34: Controller TROM class – class diagram *52*
Figure 35: Gate TROM class – Class specification *52*
Figure 36: Gate TROM class – State chart diagram *52*
Figure 37: Gate TROM class – class diagram *53*
Figure 38: SCS – Collaboration diagram *53*
Figure 39: SCS –Specification *54*
Figure 40: Set LSL trait *54*
Figure 41: TGC system class diagram *55*
Figure 42: Train TROM class – class specification *55*
Figure 43: Train TROM class – state chart diagram *56*
Figure 44: Controller TROM class – class specification *56*

Figure 45: Controller TROM class – state chart diagram *57*
Figure 46: Gate TROM class – class specification *57*
Figure 47: Gate TROM class – state chart diagram *58*
Figure 48: Sample Simulation Event List *58*
Figure 49: The Simulation Result of TGC modeling with Parameterized Events *59*
Figure 50: Robotics System Class Diagram *62*
Figure 51: User TROM class – Class specification *63*
Figure 52: User TROM class – State chart diagram *63*
Figure 53: User TROM class – Class diagram *64*
Figure 54: Vision System TROM class – Class specification *65*
Figure 55: Vision System TROM class – State chart diagram *65*
Figure 56: Vision System TROM class – Class diagram *65*
Figure 57: Belt TROM class – Class specification *66*
Figure 58: Belt TROM class – State chart diagram *66*
Figure 59: Belt TROM class – Class diagram *66*
Figure 60: Robot TROM class – Class specification *67*
Figure 61: Robot TROM class – State chart diagram *68*
Figure 62: Robot TROM class – Class diagram *68*
Figure 63: Tray TROM class – Class specification *69*
Figure 64: Tray TROM class – State chart diagram *69*
Figure 65: Tray TROM class – Class diagram *69*
Figure 66: SCS – Specification *70*
Figure 67: SCS – Collaboration diagram *70*
Figure 68: Part LSL Trait *71*
Figure 69: Queue LSL Trait *71*
Figure 70: Stack LSL Trait *71*
Figure 71: Sample Simulation Event List *71*
Figure 72: The successful simulation result of original modeling *73*
Figure 73: The failure simulation result of original modeling *74*
Figure 74: Modified Belt TROM class – Class specification *77*
Figure 75: Modified Belt TROM class – state chart diagram *77*
Figure 76: Modified Belt TROM class – class diagram *77*
Figure 77: Modified User TROM class – Class specification *78*
Figure 78: Modified User TROM class – State chart diagram *78*
Figure 79: Modified User TROM class – Class diagram *78*
Figure 80: Modified Vision System class – Class specification *79*
Figure 81: Modified Vision System class – Class diagram *79*
Figure 82: Modified Vision System class – State chart diagram *80*
Figure 83: Modified Sample Simulation Event List *80*
Figure 84: Simulation Result of Self-controlled UserModeling *81*
Figure 85: Class diagram for Robotics system *83*
Figure 86: New Belt TROM class – Class specification *84*
Figure 87: New Belt TROM class – Class diagram *84*
Figure 88: New Belt TROM class – State chart diagram *85*
Figure 89: New Vision System TROM class – Class Specification *86*
Figure 90: New Vision System TROM class – State chart diagram *86*
Figure 91: New Vision System TROM – class diagram *86*
Figure 92: New Robot TROM class – Class specification *87*
Figure 93: New Robot TROM class – Class diagram *87*
Figure 94: New Robot TROM class – State chart diagram *88*

Figure 95: SCS – New specification *88*
Figure 96: SCS – New Collaboration diagram *89*
Figure 97: Simulation Result of Buffered Belt Modeling *90*
Figure 98: The Part trait *91*
Figure 99: Robotics System class Diagram *91*
Figure 100: User TROM class – class specification *92*
Figure 101: User TROM class – state chart diagram *92*
Figure 102: User TROM class – class diagram *92*
Figure 103: Belt TROM class – class specification *93*
Figure 104: Belt TROM class – state chart diagram *93*
Figure 105: Belt TROM class – class diagram *93*
Figure 106: Vision System TROM class – class specification *94*
Figure 107: Vision System TROM class – class diagram *94*
Figure 108: Vision System TROM class – state chart diagram *95*
Figure 109: Robot TROM class – class specification *96*
Figure 110: Robot TROM class – state chart diagram *96*
Figure 111: Robot TROM class – class diagram *96*
Figure 112: Tray TROM class – class specification *97*
Figure 113: Tray TROM class – state chart specification *97*
Figure 114: Tray TROM class – class specification *97*
Figure 115: Sample Simulation Event List *97*
Figure 116: The Simulation Result of Self-controlled User modeling *98*
Figure 117: User TROM class – class specification *99*
Figure 118: User TROM class – state chart diagram *99*
Figure 119: User TROM class – class diagram *99*
Figure 120: Belt TROM class – class specification *100*
Figure 121: Belt TROM class – class diagram *100*
Figure 122: Belt TROM class – state chart diagram *101*
Figure 123: Vision System TROM class – class specification *101*
Figure 124: Vision System TROM class – class diagram *101*
Figure 125: Vision System TROM class – state chart diagram *102*
Figure 126: Robot TROM class – class specification *103*
Figure 127: Robot TROM class – state chart diagram *103*
Figure 128: Robot TROM class – class diagram *103*
Figure 129: Tray TROM class – class specification *104*
Figure 130: Tray TROM class – state chart diagram *104*
Figure 131: Tray TROM class – class diagram *104*
Figure 132: Sample Simulation Event List *104*
Figure 133: The Simulation Result of Buffered Belt modeling *106*
Figure 134: Simulation Event List 1 .... *109*
Figure 135: Simulation Event List 2 .... *109*
Figure 136: Simulation Event List 3 .... *109*
Figure 137: Simulation Event List *112*
Figure 138: Simulation Result *113*
Figure 139: The logical structure of the Validation tool *117*
Figure 140: The physical structure of the Validation tool *118*
Figure 141: The interface of the Simulator *119*
Figure 142: The interface of the Debugger *120*
Figure 143: The interface of the Query *120*
Figure 144: The interface of the Reasoning System *121*

List of Tables

Table 1: Modified grammar for transition specification *32*
Table 2: Modified grammar for SEL specification *33*
Table 3: FC and AC values of three Robotics models.. *110*

Chapter 1

Introduction

1.1 Real-Time Reactive Systems

Since their invention, computers have evolved from simple calculators to extremely complex devices, which have been entrusted with a wide range of information processing responsibilities. Alongside with information processing, interaction has become an increasingly crucial aspect of computer systems. For many types of systems, this aspect is in fact the more important one. Communication protocols, telephone switching systems and mobile robot control systems are examples of systems, which fulfill their primary goals by interacting with other systems rather than by processing information. The terms *reactive* and *real-time* systems often refer to systems of the above type.

Both reactive and interactive systems respond to environmental stimuli; however, they differ in their response mechanism. . A reactive system continuously responds to outside stimuli at the speed of the environment. A real-time reactive system is a reactive system whose responses are time-constrained. So, the system may receive messages from the environment at any time, and the system should be able to continuously respond to such stimuli within a foreseeable time frame. Examples of real-time reactive systems include air traffic control systems and nuclear reactors. In contrast, an interactive system runs at its own speed. It cannot promise to respond to an outside stimulus within a certain time frame, or even to respond at all. For example, a human computer interface application will offer users the ability to interact with the computer; however, sometimes users will have to wait for a response, during which the system may not respond to additional user inputs. Such a system is not acceptable for safety-critical systems, such as a nuclear reactor shutdown system, which must provide an instant response.

Accordingly, real-time reactive systems require a strict process to analyze the system's requirements, designs, prototypes, validation, and verification. This thesis addresses the problem raised by automatic validation of real-time reactive systems during the design phase before to implementation. We assume that such systems are modeled as timed labeled transition systems, following the TROM methodology with parameterized events. *Timed Reactive Object Model* (TROM) is a formalism introduced in [Ach95] to describe the functional and timing properties of real time reactive system using formal notations. TROMLAB framework (see Figure 1) is a developing environment built by combining object-oriented with real-time technology based on the TROM. The framework includes a number of tools to promote the rigorous development of real-time reactive systems, including the Validation tool developed to simulate the system's behavior and uncover errors in the design specification before the system is implemented. The simulated validation and the Validation tool are introduced in the following section.

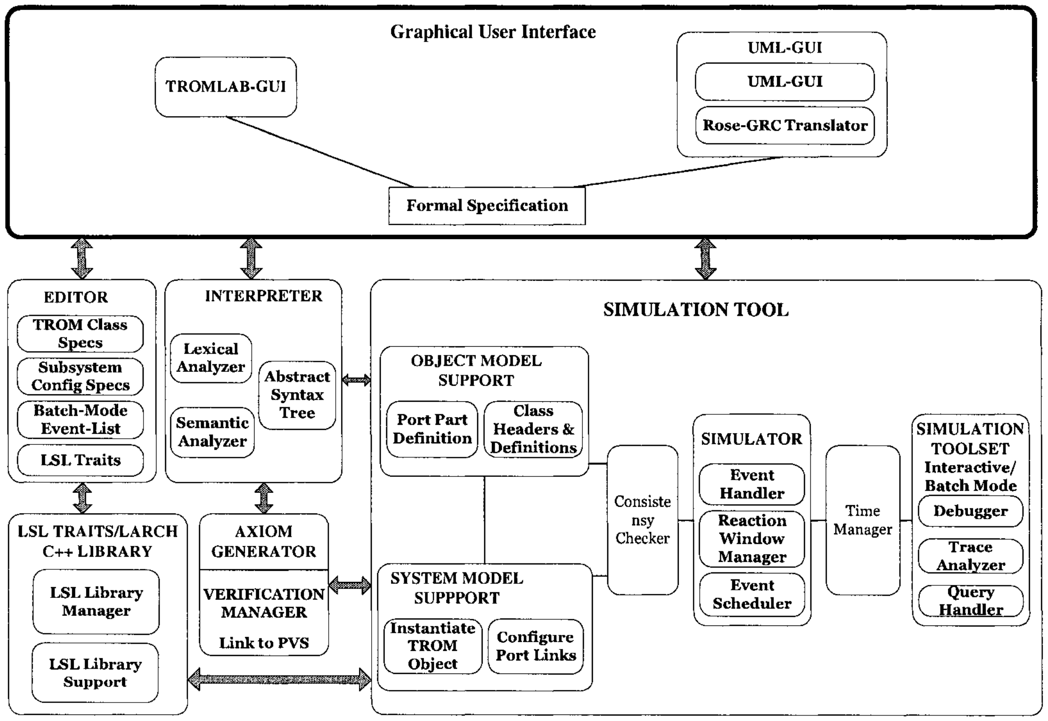


Figure 1: Architecture of TROMLAB

1.2 Simulated Validation

The development environment provides facilities for the modular design of TROM classes, the modular composition of objects to build subsystems, and analytical capabilities, which combine simulation and verification. The goal of simulated validation is to facilitate design-time debugging and design validation against the system requirements. Simulation exposes a system to a given scenario, and permits a person to observe the objects and their states, events and timing within the system. Consequently, the behavior of the system under development becomes easier to understand as the system design evolves. Trace analysis of the simulation scenarios provides invaluable insight into the behavior of the objects in the configuration, the subsystems incorporated, and the reactive system as a whole. The history of event traces allows the user to roll back the simulation clock to detect and fix faults in the design. Incorporating a reasoning system in the simulation environment also allows the use of deduction to verify properties of the system under development, based on the history of computational steps. Measurement based on the simulation analysis traces offers feedback on the system's functional complexity, and is useful to assess early system performance. The above-mentioned validation, verification and measurement facilities are integrated into one toolset.

1.3 Related Work

The validation tool for developing reactive system comes from the TROM formalism [Ach95]. The simulation improvement of this thesis is based on the simulation work of Muthiayen [Mut96] and Haidar[Hai99]. Muthiayen implemented the simulator in 1996, and later Haidar added the reasoning system to the simulator.

1.4 Major Contributions

This thesis offers the following contributions:

- It fixes flaws that existed in previous models, so that the simulator can model the behavior of real-time reactive systems successfully, like the Train-Gate-Controller and Robotics system.
- It organizes all trait-related classes into a new module – Data Model. Applying certain design patterns, the Data Model is separated from the simulator, and all data-related changes are hidden.

- May Haydar [Hay01] introduced Parameterized Events into the TROM formalism and made some modifications to the interpreter. This thesis extends the complex parameter expression, improves the interpreter, and implements the simulation for system modeling with Parameterized Events.
- This thesis also presents a new approach, based on the simulation, to assess the performance in terms of system functionality.

1.5 Thesis Outline

This thesis presents the work as follows:

- Chapter 2 briefly reviews the TROMLAB environment.
- Chapter 3 discusses some pitfalls in the simulation algorithm from previous models and proposes a new solution.
- Chapter 4 presents the current class diagram of the simulator, and the redesign of it to separate the Data Model from the simulator.
- Chapter 5 introduces the previous design of the interpreter and discusses some shortcomings in May Haydar's implementation of the interpreter. Following, it discusses the reengineering of both the interpreter and the simulator.
- Chapter 6 describes the Train-Gate-Controller case study and its modeling with Parameterized events.
- Chapter 7 presents the Robotics case study, including its remodeling with Parameterized Events.
- Chapter 8 introduces and illustrates the functionality measurement of a system based on the simulation.
- Chapter 9 offers the conclusions of our work and the research directions.

Chapter 2

TROMLAB Environment – a Brief Review of Initial Design

This chapter describes the development environment for real-time reactive systems. The TROMLAB development environment is an integrated facility based on the Timed Reactive Object Model (TROM) formalism [Ach95] for modeling, analyzing, and developing real-time reactive systems. The process model in TROMLAB supports the iterative development approach, which provides the following benefits:

- Reduces risks by exposing them early in the development process.
- Gives importance to the architecture of the system's configuration.
- Designs modules for large-scale software reuses.

The following sections present the TROM formalism and features of the TROMLAB.

2.1 TROM Formalism

The TROM formalism is a three-tier formal model illustrated in Figure 2. As a layered model, each lower tier communicates only with its immediate upper tier. The independence between the tiers makes the modularity, reuse, encapsulation, and hierarchical decomposition possible. The three-tier structure describes the system configuration, reactive classes, and relative Abstract Data Type. The upper-most tier is the subsystem configuration specification. It specifies the object definition, their collaboration, and the port links, which regulate the communication tunnels between objects. The middle tier is the TROM class, which is a Generic Reactive Class and is included in the subsystem. TROM class is a hierarchical finite state machine augmented with ports, attributes, logical assertions on the attributes, and time constraints. The lowest tier is the Larch Shared Language (LSL) trait that represents Abstract Data Type used in the TROM classes. The following sections detail the tiers.

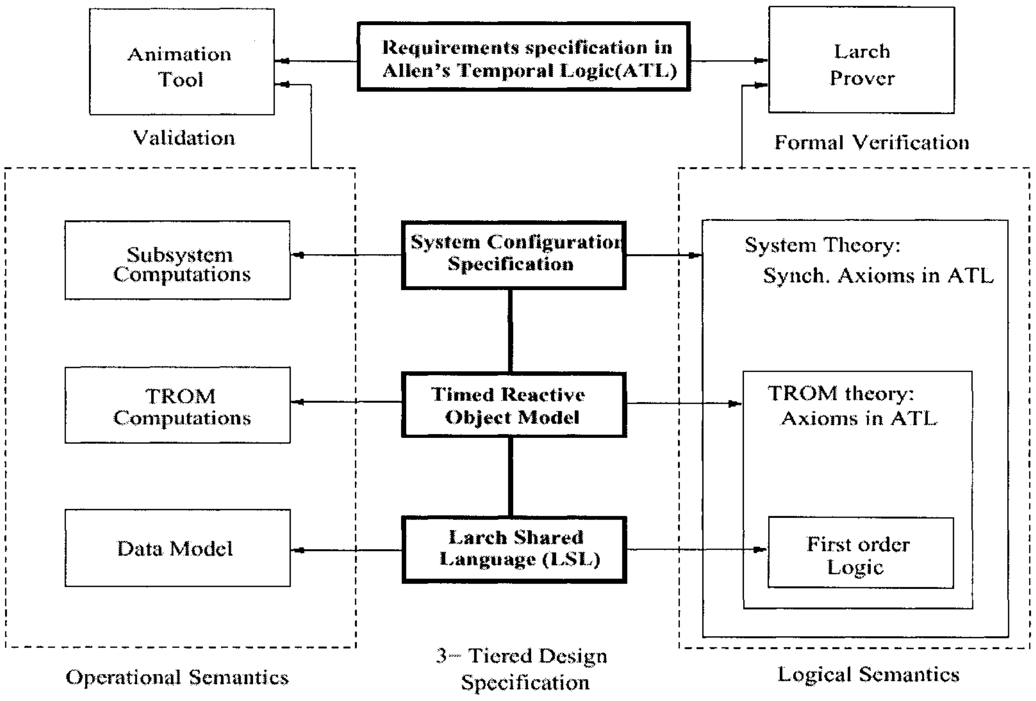

Animation Tool
Requirements specification in Allen's Temporal Logic(ATL)
Larch Prover
Validation
Formal Verification
Subsystem Computations
System Configuration Specification
System Theory: Synch. Axioms in ATL
TROM Computations
Timed Reactive Object Model
TROM theory: Axioms in ATL
Data Model
Larch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

Figure 2: Three Tiers

2.1.1 Data Abstraction Tier

The Data Abstract Tier encapsulates the Abstract Data Type. This tier uses the LSL trait to define all data types used in the middle tier except for primitive data types, such as Integer, Boolean, String, and Real etc. The Larch [GH93] provides a two-tier approach to the specification:

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

The following is a sample of an LSL trait for the Set data type:

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

Figure 3: Set trait

2.1.2 TROM Tier

A TROM models a *Generic Reactive Class* (GRC). A GRC is an augmented finite state machine with port types, attributes, hierarchical states, events triggering transitions and future events constrained by strict time intervals. A state with its attributes is an abstraction denoting environmental information or system information during a certain interval of time. A port type defines one or more ports between TROM classes for sending or receiving messages, as well as the messages passed on them. Different TROM classes need to define different port types to establish communications. Different instantiates of a port type are used to communicate with different instantiates of a TROM class. An event denotes an instantaneous signal or message. Events are classified into three types: *Input*, *Output*, and *Internal.* The Port Link synchronizes all Input and Output events, and combines with two ports ended at two TROM classes to define the communication channel between TROM classes. In addition, issue of an event may result in other events that are triggered in some strict time intervals. Thus, a GRC is a class parameterized with port types, and encapsulates behavior of all TROM objects that can be instantiated from it. In summary, a TROM has the following elements:

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

reaction is disabled when an object enters any of the disabling states of the reaction.

A formal definition of a generic reactive object, as given by Achuthan [Ach95], is an 8-tuple $(P, \mathcal{E}, \mathbf{\Theta}, X, \mathcal{L}, \mathbf{\Phi}, \mathbf{\Lambda}, \mathbf{Y})$ such that:

- P represents a finite set of port-types. A distinguished port-type is the null-type $P_\circ$ whose only port is the null port $\circ$.
- $\mathcal{E}$ represents a finite set of events and includes the silent-event tick. The set $\mathcal{E}$ - {tick} is divided into two disjoint subsets: $\mathcal{E}_{ext}$ represents the set of external events, and $\mathcal{E}_{int}$ represents the set of internal events. The set $\mathcal{E}_{in} = \{e? \mid e \in \mathcal{E}_{ext}\}$ represents the set of input events, and the set $\mathcal{E}_{out} = \{e! \mid e \in \mathcal{E}_{ext}\}$ represents the set of output events. Each $e \in (\mathcal{E}_{in} \cup \mathcal{E}_{out})$, is associated with a unique port-type $p \in P - \{P_\circ\}$.
- $\mathbf{\Theta}$ represents a finite set of states. $\theta_\circ \in \mathbf{\Theta}$, is the initial state.
- X represents a finite set of typed attributes.
- $\mathcal{L}$ represents a finite set of LSL traits introducing the abstract data types used in X.
- $\mathbf{\Phi}$ represents a function-vector$(\mathbf{\Phi}_s, \mathbf{\Phi}_{at})$ where,
  - $\mathbf{\Phi}_S$: associates with each state θ a set of states, possibly empty, called substates. A state θ is simple, if $\Phi s(\theta) = \emptyset$. By definition, the initial state $\theta_\circ$ is atomic.
  - $\mathbf{\Phi}_{at}$: associates with each state θ a set of attributes, possibly empty, called active attribute set. At each state θ, the set $\overline{\mathbf{\Phi}_{at}} = X - \mathbf{\Phi}_{at}(\theta)$ is called the dormant attribute set of θ.
- $\mathbf{\Lambda}$ represents a finite set of transition specifications including λ_{init}. A transition specification $\lambda \in \mathbf{\Lambda} - \{\lambda_{init}\}$, is a three-tuple: $< \langle \theta, \theta' \rangle \; ; e(\psi_{port}); \psi_{en} \Rightarrow \psi_{post} >$; where:
  - $\theta, \theta' \in \Theta$ are the source and destination states of the transition;

- $e(\psi_{port})$ where event e $\in \mathcal{E}$ labels the transition; ψ_{port} is an assertion on a reserved variable **pid**, where **pid** is the identifier of the port at which an interaction associated with the transition can occur. If $e \in \mathcal{E}_{int} \cup \{\textbf{tick}\}$, then the assertion ψ_{port} is absent and assumed to occur at the null port $\circ$.
- $\psi_{en} \Rightarrow \psi_{post}$, where ψ_{en} is the enabling condition and ψ_{post} is the post-condition of the transition. ψ_{en} is an assertion on the attributes in X and an assertion of the parameters in Y_d specifying the condition under which the transition is enabled. ψ_{post} is an assertion on the attributes in X and the parameters in Y_d, primed attributes in $\Phi_{at}(\theta')$ and the variable **pid** specifying the data computation associated with the transition.

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

The Figure 4 shows the anatomy of a TROM class.

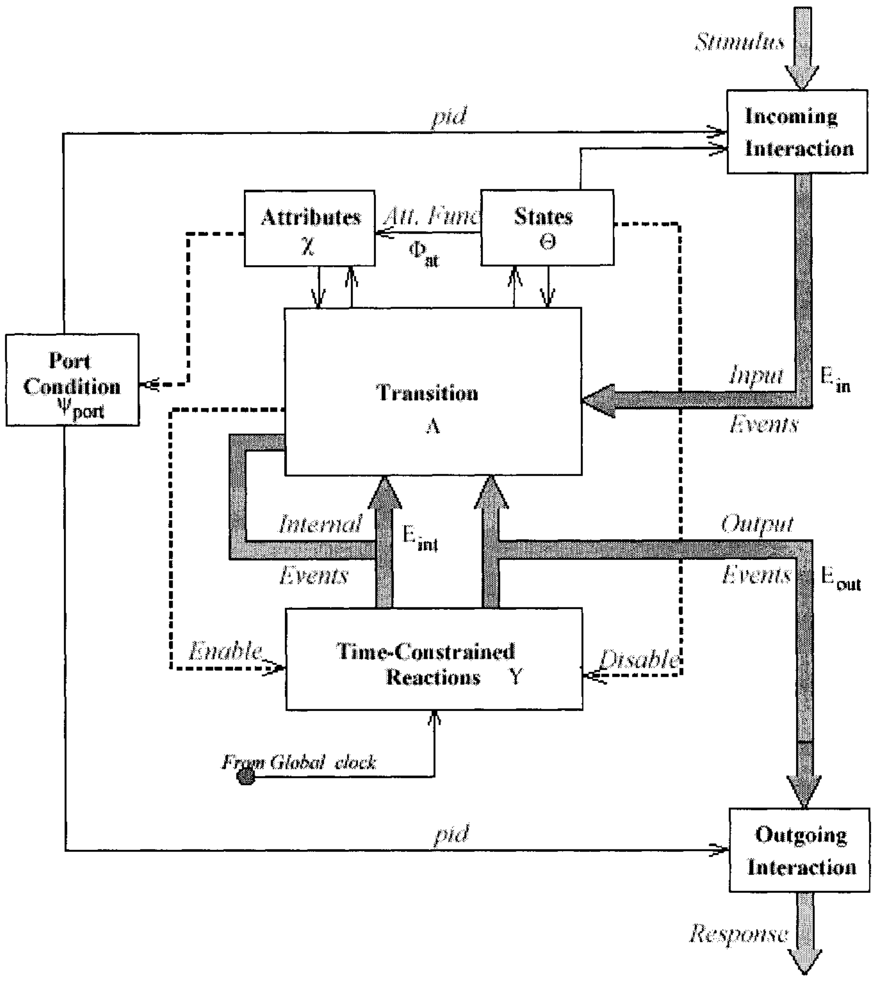


Figure 4: TROM class anatomy

2.1.3 Operational Semantics

The TROM's status captures its state at that instant. The *assignment vector* reflects the value of the TROM's attributes at that instant, and the *reaction vector* specifies the TROM's timing behavior. The *reaction vector* associates a set of *reaction windows* with each time constraint, where a *reaction window* represents an outstanding timing requirement to be satisfied by the output event or the internal event associated with the time constraint. When the *reaction vector* is null, the TROM is in a *stable status.*

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

A *computational step* [Ach95] of a TROM is an atomic step, which takes the TROM from one status to its succeeding status, as defined by the transition specifications. Every computational step of a TROM is associated with a transition in the TROM; and every transition is associated with either an interaction signal, an internal signal, or a silent signal. A computational step occurs when the TROM receives a signal and a transition specification exists that satisfies the following conditions: the triggering event for the transition causes the signal; the TROM is in the source state or a substate of the source state of the transition specification; the port-condition is satisfied if the signal is an interaction; and the assignment vector satisfies the enabling condition. These are effects of the computational step: the TROM enters the destination state; the assignment vector is modified to satisfy the post condition; and the reaction vector is modified to reflect the firing, disabling, and enabling of reactions.
A computational step causes time-constrained responses to be activated or deactivated. If the constraining event of the outstanding reaction is the event associated with the transition, and the time the event associated with the transition occurred within the reaction window of the outstanding reaction, then the reaction is fired. If the destination state of the transition associated with the computational step is a disabling state for an outstanding reaction, then the reaction is disabled. Whenever a reaction is time-constrained by the transition associated with the computational step, the reaction is enabled. The operational semantics ensures that the time cannot advance past the reaction window without either firing or disabling the associated outstanding reaction.
The following factors determines whether a TROM is *well-formed*:

- At least one transition leaves every state, thus barring the TROM from having a final terminating state.
- If more than one transition leaves a state, then the enabling conditions of the transitions should be mutually exclusive.
- Before a TROM starts executing, the values of only the active attributes in the initial state are specified; the values of the dormant attributes are undefined. An attribute will acquire a value only when it reaches the first state in which it is active. Therefore, an attribute that is dormant in the initial state must become active in some later state before the attribute is used.
- Every computational step in a TROM results in some computation of the TROM.

2.1.4 Subsystem Tier

The subsystem configuration is specified in the subsystem tier. The **Include** section indicates imported subsystems. The **Instantiate** section defines the composite objects included in the subsystem and the port instantiations owned by each object. The **Configure** section defines the port links between objects. A port link is an abstraction of a communication tunnel between two TROM objects.

2.2 SimulationValidation Components in TROMLAB

The interpreter [Sri99] and simulator [Hai99] are two components of TROMLAB for the automatic validation of real-time reactive systems. This section will briefly review them.

2.2.1 The Interpreter

Tao [Tao96] engineered the interpreter, which was the first tool implemented in TROMLAB, in C++. Haidar [Hai99] and Sriniva [Sri99] reengineered the interpreter implementation in Java. The interpreter takes the TROM specifications as inputs to check the syntax, analyze the semantics, and report error messages. Finally, it creates an Abstract Syntax Tree as the input passed into the simulator for further handling. The interpreter consists of two components and Figure 5 shows the architecture:

- **Parser**

  The parser is implemented in JavaCC and JJTree. Java Compiler Compiler is a parser generator for use with Java applications that produces Java code. JJTree is a preprocessor for JavaCC that inserts in JavaCC source actions for parse tree building. There exist separate parsers for *LSL trait*, TROM *class specification, system configuration specification (SCS)*, and *initial simulation event list.*

- **Semantics Analyzer**

  The Semantic Analzyer analyzes the class specification and subsystem configuration.

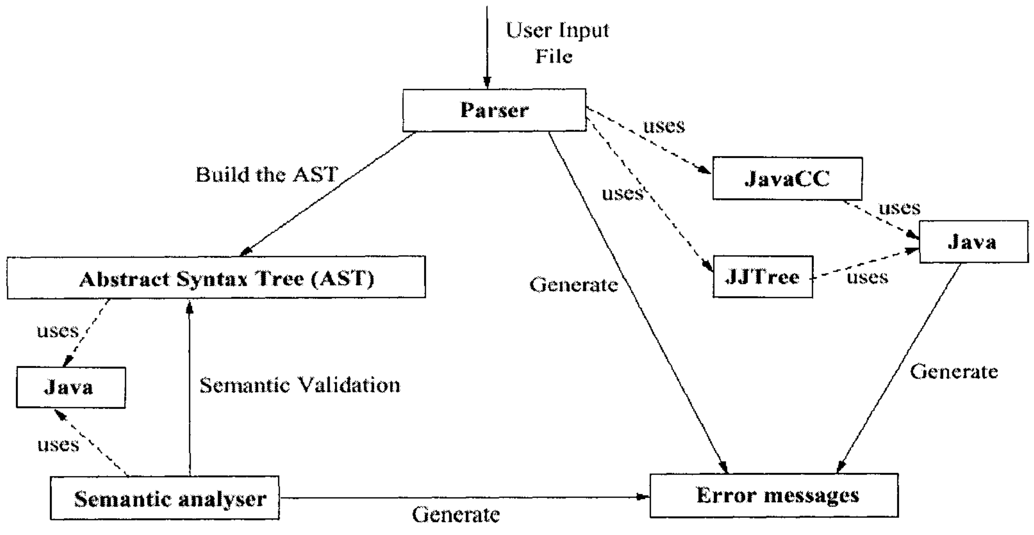


Figure 5: Architecture of interpreter

The above two tools perform the following tasks:

- *Syntactic analysis*

  Syntactic analysis ensures that all input specifications comply with the TROM grammar.

- *Semantics analysis*

  Semantics analysis performs the following jobs:

  - Verify states of a TROM have different names,
  - Verify an LSL trait is used after being declared,
  - Verify every transition has an outgoing and incoming state,
  - Verify transition specifications are well-formed logical formulas

- *Internal structure*

  A Abstract Syntax Tree is created based on the above syntax and semantic analysis.

2.2.3 The Simulation Tool

Muthiayen [Mut96] first implemented the simulation tool, which was later reengineered by Haidar [Hai99] with Java. Figure 6 illustrates the simulator architecture, which has two working modes:

- *Debugger mode*: In this mode, the simulator runs step-by-step. With the debugger, users may query run-time variables, and roll back as well as inject new events at each step.
- *Normal mode:* The simulator runs uninterrupted in this mode until it reaches a stable state. Whether the result is or is not correct then depends on the correctness of the specification.

The Simulation tool consists of the following components:

- *Consistency checker*: This ensures the continuous flow of interactions by checking deadlocked configurations.
- *Simulator*: This consists of an event handler, a reaction window manager, and an event scheduler.
  - The event handler handles events according to the transition definition.
  - The reaction window manager activates the computational step, which fires, disables or enables the transition-causing events.
  - The event scheduler schedules an event and causes it to occur at a random time within the corresponding reaction window.
- *Validation tool:* This consists of a debugger, a trace analyzer, and a query handler.
  - The debugger allows the user to to analyze the running status of the evaluated system. The user can also inject new events and roll back the system simulation.
  - The trace analyzer allows the user to analyze of the simulation scenario. It gives feedback on the evolution of the status of the objects in the system, and the outcome of the simulation event.
  - The query handler allows the user to examine the data in the AST for the TROM class to which the object belongs, and supporting analysis of the static components during simulation.

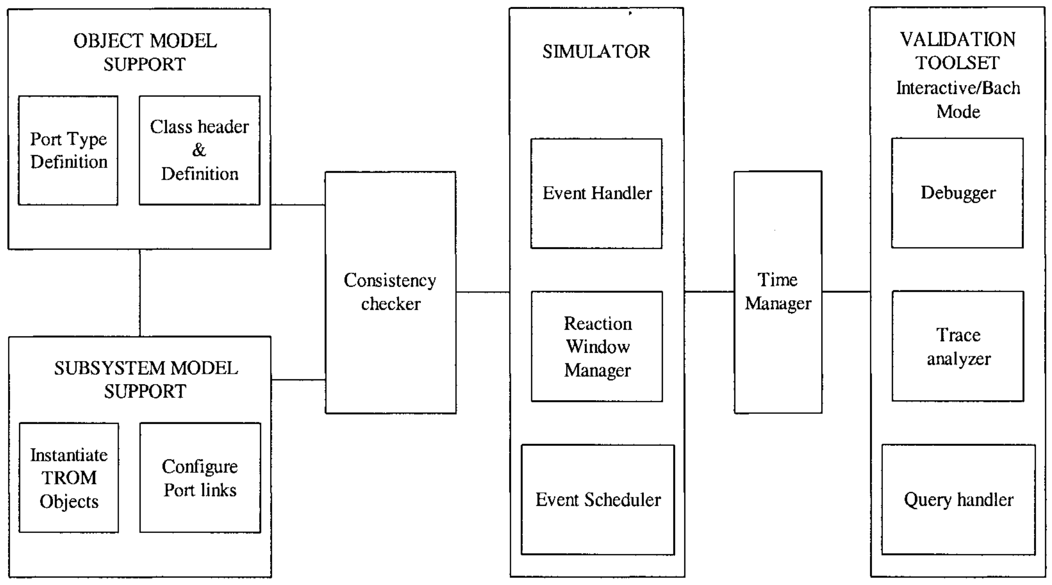


Figure 6: Architecture of Validation tool

One of the goals of this thesis is to correct flaws related to time conflicts in the previous simulation model. The following chapter introduces the Time Interval Comparison algorithm and illustrates it correctness on a Train-Gate-Controller case study.

Chapter 3

Time Interval Comparison Algorithm

3.1 Motivation

Muthiayen [Mut96] designed and implemented the previous validation tool in C++. Haidar [Hai99] and Sriniva [Sri99] reengineered it in Java and added reasoning capabilities. Haydar [Hay01] then introduced parameterized events to the TROM formalism. However, the previous simulation algorithm contained a flaw that prevented the successful simulation of a system's behavior. The following sections will analyze the above-mentioned problem in the simulation results of the Train-Gate-Controller case study. Next, we will present an algorithm for the time interval comparison, which solves the above-mentioned problem. Finally, we will illustrate the improved simulation algorithm on the case study.
The inputs into the validation tool are the Train-Gate-Controller formal model and the initial simulation event list, as described below.

3.2 Train-Gate-Controller

In the non-parameterized version of Train-Gate-Controller model (See Section 6.2), three trains pass through two gates coordinated by two controllers. In the system, the trains communicate with the controllers by sending messages, and the controllers control the gates by issuing commands. When a train approaches a gate, it sends the message ***Near*** to the controller of that gate. The controller then instructs the gate to close. The controller commands the gate to open after it receives a message from the train that has departed. Section 6.2 describes the case study formal model (TROM classes specifications, LSL traits and system configuration specification (SCS)).

3.3 Sample Simulation Event List

The event list provides a timeline of external stimuli used in the Train-Gate-Controller simulation, and consists of six ***Near*** events involving train objects that are instantiated in the SCS. These events simulate a scenario involving two trains passing through two gates sequentially, while a third train passes through these same two gates in the opposing direction.

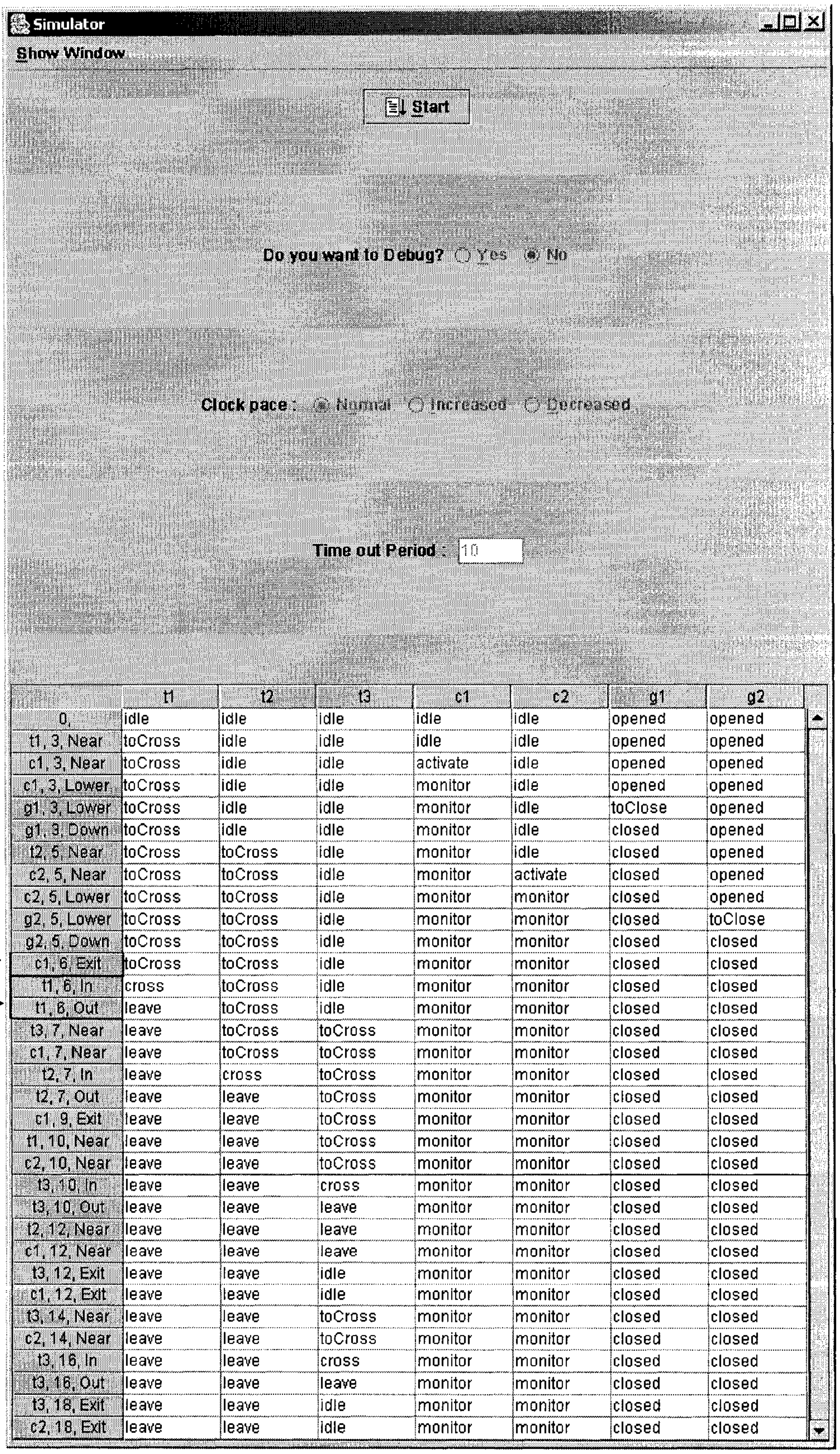
```
SEL: TCG
   Near, t1, @C1, 3;
   Near, t2, @C2, 5;
   Near, t3, @C1, 7;
   Near, t1, @C2, 10;
   Near, t2, @C1, 12;
   Near, t3, @C2, 14;
end
```

Figure 7: Sample Simulation Event List

3.4 Simulation Result of the Present Simulator

Figure 8 provides the simulation results from the TGC system. In the output table, the first column displays the time during which an object (i.e., train, gate or controller) sends or receives a signal, followed by the event type, with "!" denoting an output event, "?" denoting an input event, and no symbol denoting an internal event. The remaining columns, starting from column t1 moving left to right, show the state of train 1 (t1), train 2 (t2), train 3(t3), controller 1 (c1), controller 2 (c2), gate (g1), and gate 2 (g2), respectively, at the time of each event. For example, the first row shows that at 0 time unit – the initial state of the system– t1, t2, t3, c1, and c2 are in an idle state, while g1 and g2 are in an opened state. The second row shows that t1 enters a toCross state after t1 sends an output signal at the 3rd time unit

The following section will analyze the flaws in the algorithm on the simulation result exposed in figure 8.



Do you want to Debug? Yes [✓ No]

Clock pace : [✓ Normal] Increased Decreased

Time out Period : 10

	t1	t2	t3	c1	c2	g1	g2
0,	idle	idle	idle	idle	idle	opened	opened
t1, 3, Near	toCross	idle	idle	idle	idle	opened	opened
c1, 3, Near	toCross	idle	idle	activate	idle	opened	opened
c1, 3, Lower	toCross	idle	idle	monitor	idle	opened	opened
g1, 3, Lower	toCross	idle	idle	monitor	idle	toClose	opened
g1, 3, Down	toCross	idle	idle	monitor	idle	closed	opened
t2, 5, Near	toCross	toCross	idle	monitor	idle	closed	opened
c2, 5, Near	toCross	toCross	idle	monitor	activate	closed	opened
c2, 5, Lower	toCross	toCross	idle	monitor	monitor	closed	opened
g2, 5, Lower	toCross	toCross	idle	monitor	monitor	closed	toClose
g2, 5, Down	toCross	toCross	idle	monitor	monitor	closed	closed
c1, 6, Exit	toCross	toCross	idle	monitor	monitor	closed	closed
t1, 6, In	cross	toCross	idle	monitor	monitor	closed	closed
t1, 6, Out	leave	toCross	idle	monitor	monitor	closed	closed
t3, 7, Near	leave	toCross	toCross	monitor	monitor	closed	closed
c1, 7, Near	leave	toCross	toCross	monitor	monitor	closed	closed
t2, 7, In	leave	cross	toCross	monitor	monitor	closed	closed
t2, 7, Out	leave	leave	toCross	monitor	monitor	closed	closed
c1, 9, Exit	leave	leave	toCross	monitor	monitor	closed	closed
t1, 10, Near	leave	leave	toCross	monitor	monitor	closed	closed
c2, 10, Near	leave	leave	toCross	monitor	monitor	closed	closed
t3, 10, In	leave	leave	cross	monitor	monitor	closed	closed
t3, 10, Out	leave	leave	leave	monitor	monitor	closed	closed
t2, 12, Near	leave	leave	leave	monitor	monitor	closed	closed
c1, 12, Near	leave	leave	leave	monitor	monitor	closed	closed
t3, 12, Exit	leave	leave	idle	monitor	monitor	closed	closed
c1, 12, Exit	leave	leave	idle	monitor	monitor	closed	closed
t3, 14, Near	leave	leave	toCross	monitor	monitor	closed	closed
c2, 14, Near	leave	leave	toCross	monitor	monitor	closed	closed
t3, 16, In	leave	leave	cross	monitor	monitor	closed	closed
t3, 16, Out	leave	leave	leave	monitor	monitor	closed	closed
t3, 18, Exit	leave	leave	idle	monitor	monitor	closed	closed
c2, 18, Exit	leave	leave	idle	monitor	monitor	closed	closed

Figure 8: The simulating result of present Simulator

3.4 Problem Analysis

In the event indicated by the first black arrow, controller 1 received an *Exit* signal at the 6th time unit. According to the Figure 31, train t1 should have sent this signalto controller 1 at the 6th time unit. But, the *Exit* signal from t1 disappeared from the results, meaning that train t1 did not issue the *Exit* signal because its pre-condition or port-condition was unsatisfied. Here, the pre-condition of transition R3 of Train TROM class is unsatisfied because the current state of t1 is "toCross," not "leave."
The area marked by the second black arrow includes two events – Following the Exit event, train t1 issues an *In* and an *Out* signals at the 6th time unit. These events themselves have no problems, but the order of the triggering events – *Exit*, *In*, and *Out* – are incorrect. The right order should be *In*, *Out*, and *Exit*, as presented in the Figure 30. This problem derives from a flaw in the current simulation algorithm. When an event triggers a transition, the simulator will check whether a time constraint is associated with this transition. If an event that is time-constrained by this transition exists, the simulator will schedule this event based only on the period defined by the event's time interval, i.e. the simulator randomly generates a triggering time for this event from the event's time interval without considering other factors. Therefore, train t1 may issue an *Exit* signal before it issues an *In* signal because their time intervals overlap. As long as the generated occurrence time for the *Exit* event is later than that for the *In* event, this problem will occur. Therefore, it is necessary to find a way to regulate the event triggering order in addition to their triggering time. The following presented algorithm is used to reach this goal.

3.5 Time Interval Comparison Algorithm

There are two kinds of transitions: concurrent and sequential. Concurrent transitions are those transitions that are issued from same source states. These are further divided into two types: radical and parallel. Radical transitions have different destination states, while parallel transitions have the same destination states. In contrast, the sequential transitions have different source states.
In Page 24 [Hai99], Haidar stated that a well-formed TROM should have the following properties: "If there is more than one transition leaving a state, then the enabling conditions of transitions should be mutually exclusive." Therefore, the

Simulator will evaluate the enabling conditions of concurrent transitions to determine which event will be issued.

The *Time Interval Comparison Algorithm* focuses just on the sequential transitions and depends on the time interval of the constrained events to determine their order. To do this, we distinguish an event's time intervals with different cases and set a *Base Time* for each TROM object. The *Base Time* records the triggering time of the last-issued event of the TROM object. Then, the next constrained event will be triggered sometime during interval between the upper bound of the time interval and either the base time or the lower bound of the time interval, which ever occurs later. For distinguishing all kinds of time interval cases, we borrow from Allen's logic [All84] the predicates, which also are used by Achuthan [Ach95], to express temporal relationships between time intervals. There are seven temporal relationships between time intervals: *Before*, *Meet*, *Overlaps*, *Equal*, *During*, *Starts*, and *Finishes*. We will discuss each of them in the next subsections.

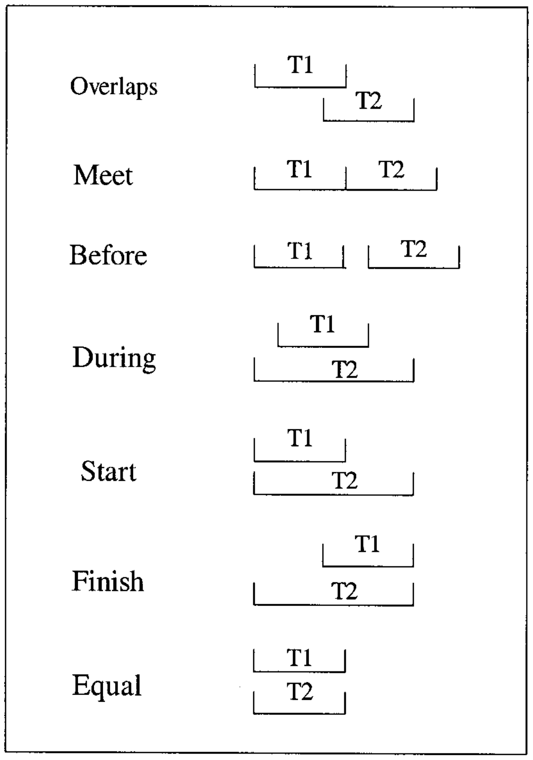


Figure 9: Allen's temporal predicates

O(T) = $\{x \mid u \le x \le v, T = [u, v]\}$, variable O means time during which an event occurs

$T1 = [u1, v1], v1 > u1$

$T2 = [u2, v2], v2 > u2$

T1 denotes the time interval of the event that should occur first, and T2 stands for the time interval of the subsequent constrained event.

3.5.1 Overlaps

$\text{Overlaps}(T1, T2) \triangleq u1 < u2 < v1 < v2$

Comparing upper bound: An event with a smaller upper bound will occur before that of the event with larger upper bound.

3.5.2 Meet

$\text{Meet}(T1, T2) \triangleq v1 = u2$

Comparing upper bound: An event with a smaller upper bound will occur before that of the event with larger upper bound.

3.5.3 Before

$\text{Before}(T1,T2) \triangleq v1 < u2$

Comparing upper bound: An event with a smaller upper bound will occur before that of the event with larger upper bound.

3.5.4 During

$\text{During}(T1,T2) \triangleq u2 < u1 \wedge v1 < v2$

Comparing upper bound: An event with a smaller upper bound will occur before that of the event with larger upper bound.

3.5.5 Start

$\text{Start}(T1,T2) \triangleq u2 = u1 \wedge v1 < v2$

Comparing upper bound: An event with a smaller upper bound will occur before that of the event with larger upper bound.

3.5.6 Finish

Finish(T1,T2) $\triangleq$ u1 < u2 $\wedge$ v1 = v2

Comparing lower bound: An event with a smaller lower bound will occur before that of the event with larger lower bound.

3.5.7 Equal

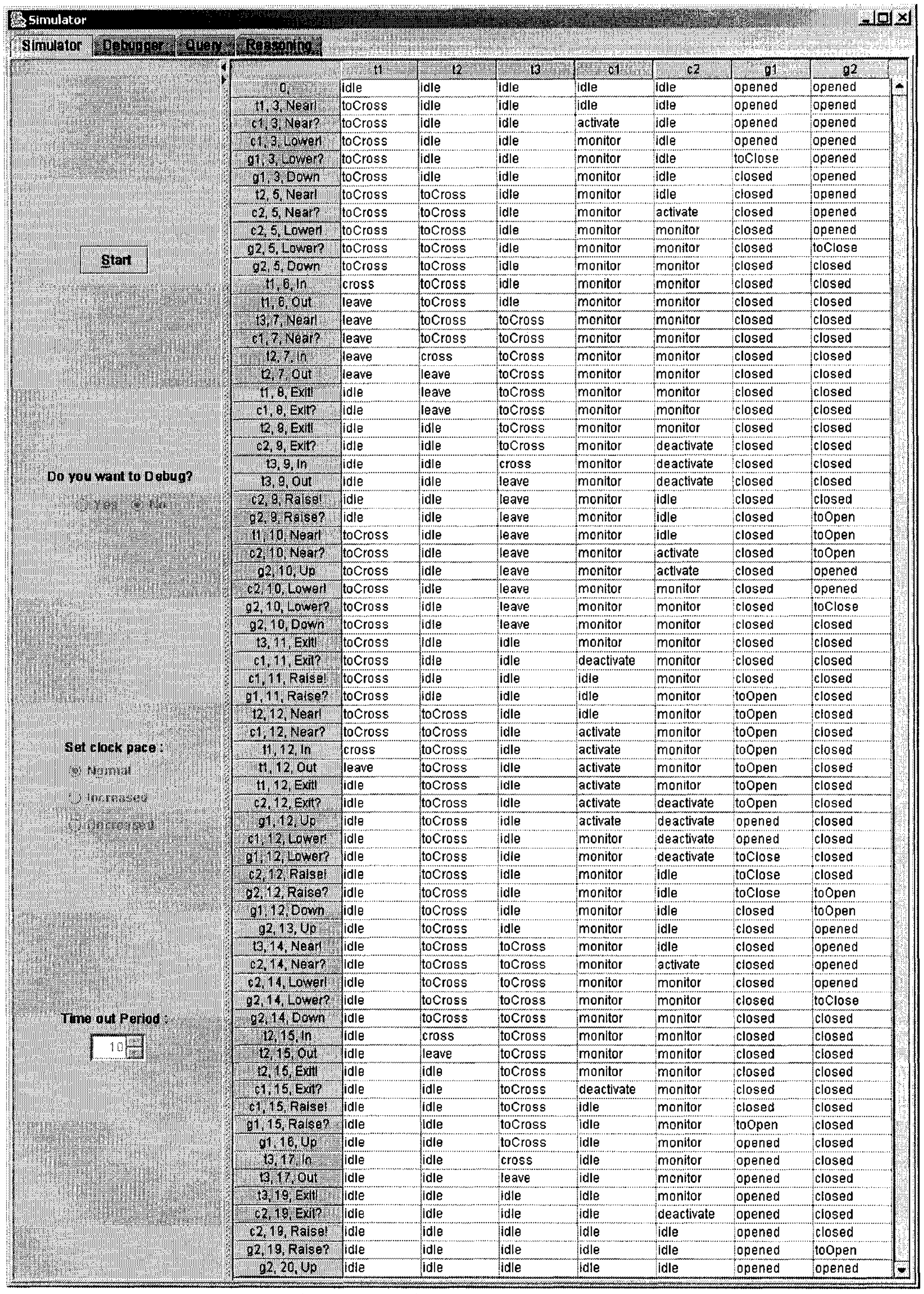
Equal(T1,T2) $\triangleq$ u2 = u1 $\wedge$ v1 = v2

When transitions occur concurrently, the simulator will evaluate the enabling conditions of the concurrent transitions to determine which events will be issued in our system. Otherwise, the translator should avoid this case by transforming it into a different temporal relationship.

With this new algorithm, the validation tool can now handle the TGC system.

3.5.8 Simulation Result with Modified Algorithm

The Figure 10 displays the simulation results after the modified algorithm is implemented in the simulator. After the system converges to a stable state, all objects go to their initial states. The successful simulation results demonstrate that the modified algorithm iscorrect.



Simulator

Simulator | Debugger | Query | Reasoning

Start

Do you want to Debug?

Yes [✓ No]

Set clock pace :

[✓ Normal]

Increased

Decreased

Time out Period : 10

	t1	t2	t3	c1	c2	g1	g2
0,	idle	idle	idle	idle	idle	opened	opened
t1, 3, Near!	toCross	idle	idle	idle	idle	opened	opened
c1, 3, Near?	toCross	idle	idle	activate	idle	opened	opened
c1, 3, Lower!	toCross	idle	idle	monitor	idle	opened	opened
g1, 3, Lower?	toCross	idle	idle	monitor	idle	toClose	opened
g1, 3, Down	toCross	idle	idle	monitor	idle	closed	opened
t2, 5, Near!	toCross	toCross	idle	monitor	idle	closed	opened
c2, 5, Near?	toCross	toCross	idle	monitor	activate	closed	opened
c2, 5, Lower!	toCross	toCross	idle	monitor	monitor	closed	opened
g2, 5, Lower?	toCross	toCross	idle	monitor	monitor	closed	toClose
g2, 5, Down	toCross	toCross	idle	monitor	monitor	closed	closed
t1, 6, In	cross	toCross	idle	monitor	monitor	closed	closed
t1, 6, Out	leave	toCross	idle	monitor	monitor	closed	closed
t3, 7, Near!	leave	toCross	toCross	monitor	monitor	closed	closed
c1, 7, Near?	leave	toCross	toCross	monitor	monitor	closed	closed
t2, 7, In	leave	cross	toCross	monitor	monitor	closed	closed
t2, 7, Out	leave	leave	toCross	monitor	monitor	closed	closed
t1, 8, Exit!	idle	leave	toCross	monitor	monitor	closed	closed
c1, 8, Exit?	idle	leave	toCross	monitor	monitor	closed	closed
t2, 9, Exit!	idle	idle	toCross	monitor	monitor	closed	closed
c2, 9, Exit?	idle	idle	toCross	monitor	deactivate	closed	closed
t3, 9, In	idle	idle	cross	monitor	deactivate	closed	closed
t3, 9, Out	idle	idle	leave	monitor	deactivate	closed	closed
c2, 9, Raise!	idle	idle	leave	monitor	idle	closed	closed
g2, 9, Raise?	idle	idle	leave	monitor	idle	closed	toOpen
t1, 10, Near!	toCross	idle	leave	monitor	idle	closed	toOpen
c2, 10, Near?	toCross	idle	leave	monitor	activate	closed	toOpen
g2, 10, Up	toCross	idle	leave	monitor	activate	closed	opened
c2, 10, Lower!	toCross	idle	leave	monitor	monitor	closed	opened
g2, 10, Lower?	toCross	idle	leave	monitor	monitor	closed	toClose
g2, 10, Down	toCross	idle	leave	monitor	monitor	closed	closed
t3, 11, Exit!	toCross	idle	idle	monitor	monitor	closed	closed
c1, 11, Exit?	toCross	idle	idle	deactivate	monitor	closed	closed
c1, 11, Raise!	toCross	idle	idle	idle	monitor	closed	closed
g1, 11, Raise?	toCross	idle	idle	idle	monitor	toOpen	closed
t2, 12, Near!	toCross	toCross	idle	idle	monitor	toOpen	closed
c1, 12, Near?	toCross	toCross	idle	activate	monitor	toOpen	closed
t1, 12, In	cross	toCross	idle	activate	monitor	toOpen	closed
t1, 12, Out	leave	toCross	idle	activate	monitor	toOpen	closed
t1, 12, Exit!	idle	toCross	idle	activate	monitor	toOpen	closed
c2, 12, Exit?	idle	toCross	idle	activate	deactivate	toOpen	closed
g1, 12, Up	idle	toCross	idle	activate	deactivate	opened	closed
c1, 12, Lower!	idle	toCross	idle	monitor	deactivate	opened	closed
g1, 12, Lower?	idle	toCross	idle	monitor	deactivate	toClose	closed
c2, 12, Raise!	idle	toCross	idle	monitor	idle	toClose	closed
g2, 12, Raise?	idle	toCross	idle	monitor	idle	toClose	toOpen
g1, 12, Down	idle	toCross	idle	monitor	idle	closed	toOpen
g2, 13, Up	idle	toCross	idle	monitor	idle	closed	opened
t3, 14, Near!	idle	toCross	toCross	monitor	idle	closed	opened
c2, 14, Near?	idle	toCross	toCross	monitor	activate	closed	opened
c2, 14, Lower!	idle	toCross	toCross	monitor	monitor	closed	opened
g2, 14, Lower?	idle	toCross	toCross	monitor	monitor	closed	toClose
g2, 14, Down	idle	toCross	toCross	monitor	monitor	closed	closed
t2, 15, In	idle	cross	toCross	monitor	monitor	closed	closed
t2, 15, Out	idle	leave	toCross	monitor	monitor	closed	closed
t2, 15, Exit!	idle	idle	toCross	monitor	monitor	closed	closed
c1, 15, Exit?	idle	idle	toCross	deactivate	monitor	closed	closed
c1, 15, Raise!	idle	idle	toCross	idle	monitor	closed	closed
g1, 15, Raise?	idle	idle	toCross	idle	monitor	toOpen	closed
g1, 16, Up	idle	idle	toCross	idle	monitor	opened	closed
t3, 17, In	idle	idle	cross	idle	monitor	opened	closed
t3, 17, Out	idle	idle	leave	idle	monitor	opened	closed
t3, 19, Exit!	idle	idle	idle	idle	monitor	opened	closed
c2, 19, Exit?	idle	idle	idle	idle	deactivate	opened	closed
c2, 19, Raise!	idle	idle	idle	idle	idle	opened	closed
g2, 19, Raise?	idle	idle	idle	idle	idle	opened	toOpen
g2, 20, Up	idle	idle	idle	idle	idle	opened	opened

Figure 10: Simulating result with modified simulation algorithm

By fixing the flaws of the simulator, we have developed a workable simulation system. However, a good system should not only work, but also have an extensible structure. So, the next chapter will describe the separation of the data-related classes from the simulation algorithm-related classes.

Chapter 4

Data Model Separation

In the three-tier structure of the object-oriented methodology introduced by Achuthan [Ach95], the Data Model is the lowest tier in the TROM model and defines all abstract data types throughout the TROMLAB. In the previous implementation, all LSL trait-related classes were implemented in the simulator, thereby reducing the simulator's flexibility in handling case studies with different abstract data types. This chapter explains the separation of the data model from the simulation algorithm-related classes.

4.1 Motivation

Data-related classes coupled cohesively with the simulation algorithm implementation classes, which are irrelative to traits handling. If the system required the introduction of new data types for future extensions, the whole simulator would have to be analyzed and changed carefully. The maintenance and error checking would also be difficult. Therefore, if the trait-related classes were extracted from the interpreter and the simulator modules, we would obtain the following benefits:

(1) Isolation of the trait-related changes from other modules,

(2) Increased extensibility for introducing new data types.

4.2 Previous Design

In the previous design, all LSL trait-related classes were implemented in simulator. The *LSLLibraryManager* is the factory class for LSL trait object creation. The *LSLLibrarySupport* provides the function calls defined in LSL trait classes. The *ObjectModelSupport* evaluates logical assertions included in transition specifications. The *LSLLibraryManager* and the *LSLLibrarySupport* classes interact with various trait classes directly. In each simulation computational step, the *ObjectModelSupport* will be invoked to evaluate the port-condition, pre-condition, and post-condition. Meanwhile, it will invoke the *LSLLibrarySupport* class to evaluate expressions.

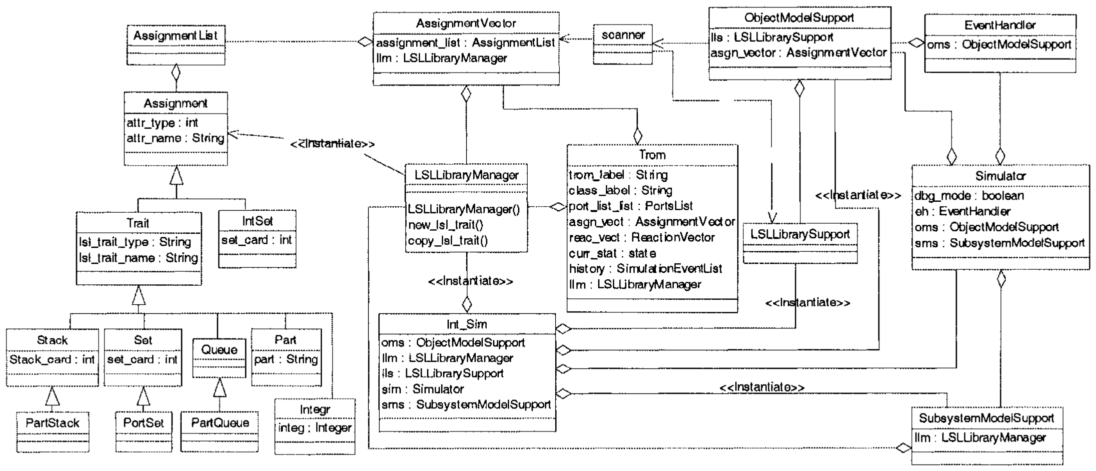


Figure 11: The relation between Data-related classes in previous design

4.3 A Design Pattern – Façade

The Façade pattern [GOF94] customizes a unified interface for a subsystem. Therefore, it isolates the outside from changes in the subsystem.

The goal of Facade pattern is to reduce the dependencies between subsystems by defining a higher-level component that contains and centralizes complex interactions between lower-level components. It also decouples lower-level components from one another, making designs more flexible and comprehensible. Generally, It works by providing an additional reusable object that hides most of the complexity of working with the other classes from client classes.

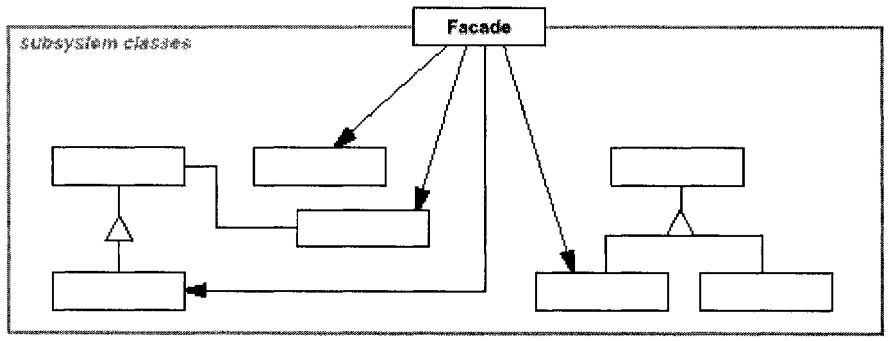


Figure 12: Façade pattern

4.4 New Design

We designed a unified interface *IDataSupport* realized by a class *DataSupport*. The interface defines all functions that the DataModel subsystem provides to the outside. The class *DataSupport* just implements the functions by transferring messages to corresponding functions of the behind-the-scene classes. Accordingly, as long as the interface isn't changed, the behind-the-scene classes can be changed arbitrarily without affecting the outside, and even the whole implementation of the interface *IDataSupport* can be replaced by a different class to change the behaviors of the subsystem completely. By changing the behind-the-scene classes, we can change the behavior of the system without touching the outside classes.

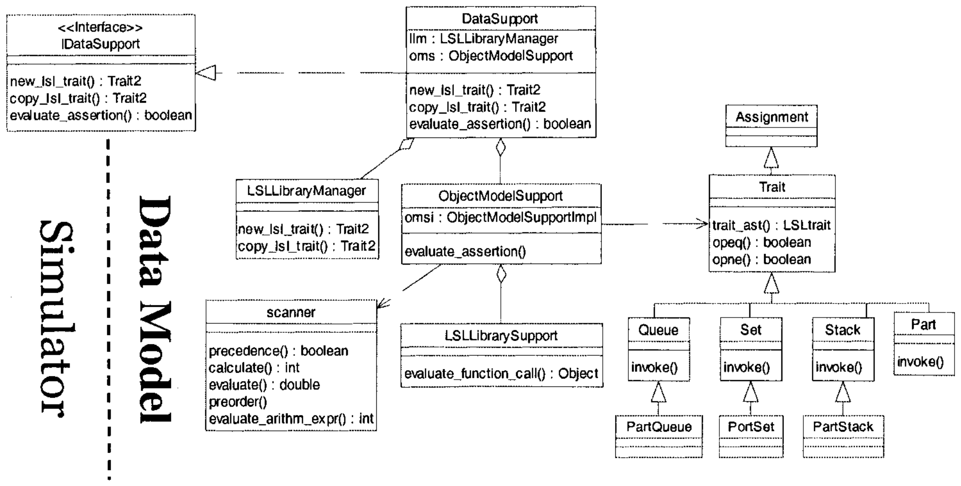


Figure 13: New design with Façade pattern

Separating the data model from the simulator results in the following benefits:

(1) Decouples the relationship between DataModel and IntSim modules;

(2) Makes the interfacebetween classes more manageable;

(3) Hides all changes related to data behind the interface of the DataModel module;

(4) High extensibility.

We have successfully tested the improved, well-structured simulator on the Train-Gate-Controller. Chapter 5 discusses the extension of the validation tool's capabilities by implementing the Parameterized Events grammar.

Chapter 5

Parameterized Events

This Chapter reviews the parameterized events introduced by Haydar [Hay01] to TROM formalism, and extends the syntax of the parameterized events to the simulation event list. Later on, it revises the interpreter and simulator to accommodate the new syntax.

5.1 Review the Parameterized Events

Some large applications have complex state machines with many states and transitions. In the transitions of the state machines, some similar transitions with same events only differ in the values to be exchanged between the components affected by them. Using parameters in these events is like using function parameters. A parameterized event represents a family of events that are distinguished by the values that can be assigned to its parameters.

Introducing parameterized events to TROM will derive the following benefits:

- Parameters reduce the complexity of comprehension and avoid the state explosion of a system.
- Parameters add expressiveness to TROM formalism.
- Parameters may transport data between objects in a subsystem.

5.1.1 Syntax of Parameterized Events

The formal definition of TROM defined by Achuthan [Ach95] and extended to cover parameterized events by Haydar [Hay01], is as follows:

A generic reactive object is an 9-tuple ($\mathcal{P}$, $\mathcal{E}$, $\mathbf{\Theta}$, $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{L}$, $\mathbf{\Phi}$, Λ, Υ) such that:

- $\mathcal{P}$ represents a finite set of port-types. A distinguished port-type is the null-type $P_\circ$ whose only port is the null port $\circ$.
- $\mathcal{E}$ represents a finite set of events and includes the silent-event tick. The set $\mathcal{E}$ - {tick} is divided into two disjoint subsets: $\mathcal{E}_{ext}$ represents the set of external events, and $\mathcal{E}_{int}$ represents the set of internal events. The set $\mathcal{E}_{in} = \{e? \mid e \in \mathcal{E}_{ext}\}$ represents the set of input events, and the set $\mathcal{E}_{out} = \{e! \mid e \in \mathcal{E}_{ext}\}$ represents the set of output events. Each $e \in (\mathcal{E}_{in} \cup \mathcal{E}_{out})$, is associated with a unique port-type $p \in \mathcal{P} - \{P_\circ\}$.
- $\mathbf{\Theta}$ represents a finite set of states. $\theta_\circ \in \mathbf{\Theta}$, is the initial state.
- $\mathcal{X}$ represents a finite set of typed attributes.
- $\mathcal{Y}$ represents a finite set of typed parameters. The set $\mathcal{Y} = \mathcal{Y}_d \cup \mathcal{Y}_s$, which $\mathcal{Y}_d$ is a set of dynamic parameters, and $\mathcal{Y}_s$ is a set of static parameters.
- $\mathcal{L}$ represents a finite set of LSL traits introducing the abstract data types used in $\mathcal{X}$.
- $\mathbf{\Phi}$ represents a function-vector ($\mathbf{\Phi}_s$, $\mathbf{\Phi}_{at}$, $\mathbf{\Phi}_p$) where,
  - $\mathbf{\Phi}_s$: associates with each state θ a set of states, possibly empty, called substates. A state θ is simple, if $\mathbf{\Phi}_s(\theta) = \emptyset$. By definition, the initial state $\theta_\circ$ is atomic.
  - $\mathbf{\Phi}_{at}$: associates with each state θ a set of attributes, possibly empty, called active attribute set. At each state θ, the set $\overline{\mathbf{\Phi}_{at}} = \mathcal{X} - \mathbf{\Phi}_{at}(\theta)$ is called the dormant attribute set of θ.
  - $\mathbf{\Phi}_p$: associates with each state θ a set of parameters, possibly empty, called active parameter set. At each state θ, the set $\overline{\mathbf{\Phi}_p} = \mathcal{Y} - \mathbf{\Phi}_p(\theta)$ is called the dormant parameter set of θ.

- Λ represents a finite set of transition specifications including λ_{init}. A transition specification $\lambda \in \Lambda - \{\lambda_{init}\}$, is a three-tuple:

  $$< \langle \theta, \theta' \rangle \ ; e[\psi_{parm}](\psi_{port});\ \psi_{en} \Rightarrow \psi_{post} >$$

  where:

  - $\theta, \theta' \in \Theta$ are the source and destination states of the transition;
  - $e[\psi_{parm}](\psi_{port})$ where event $e \in \mathcal{E}$ labels the transition; ψ_{parm} is an assertion on the parameters in Y_d that can be absent if Y_d is empty; ψ_{port} is an assertion on a reserved variable **pid**. **pid** is the identifier of the port at which an interaction associated with the transition can occur. If $e \in \mathcal{E}_{int} \cup \{\textbf{tick}\}$, then the assertion ψ_{port} is absent and assumed to occur at the null port $\circ$.
  - $\psi_{en} \Rightarrow \psi_{post}$, where ψ_{en} represents the enabling condition and ψ_{post} represents the post-condition of the transition. ψ_{en} represents an assertion on the attributes in X and an assertion of the parameters in Y_d specifying the condition under which the transition is enabled. ψ_{post} represents an assertion on the attributes in X and the parameters in Y_d, primed attributes in $\Phi_{at}(\theta')$ and the variable **pid** specifying the data computation associated with the transition.

- **Y** represents a finite set of time-constraints of the form $(\lambda_i, e_i', [l, u], \Theta_i)$ where:

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

The modified anatomy of TROM class is show below:

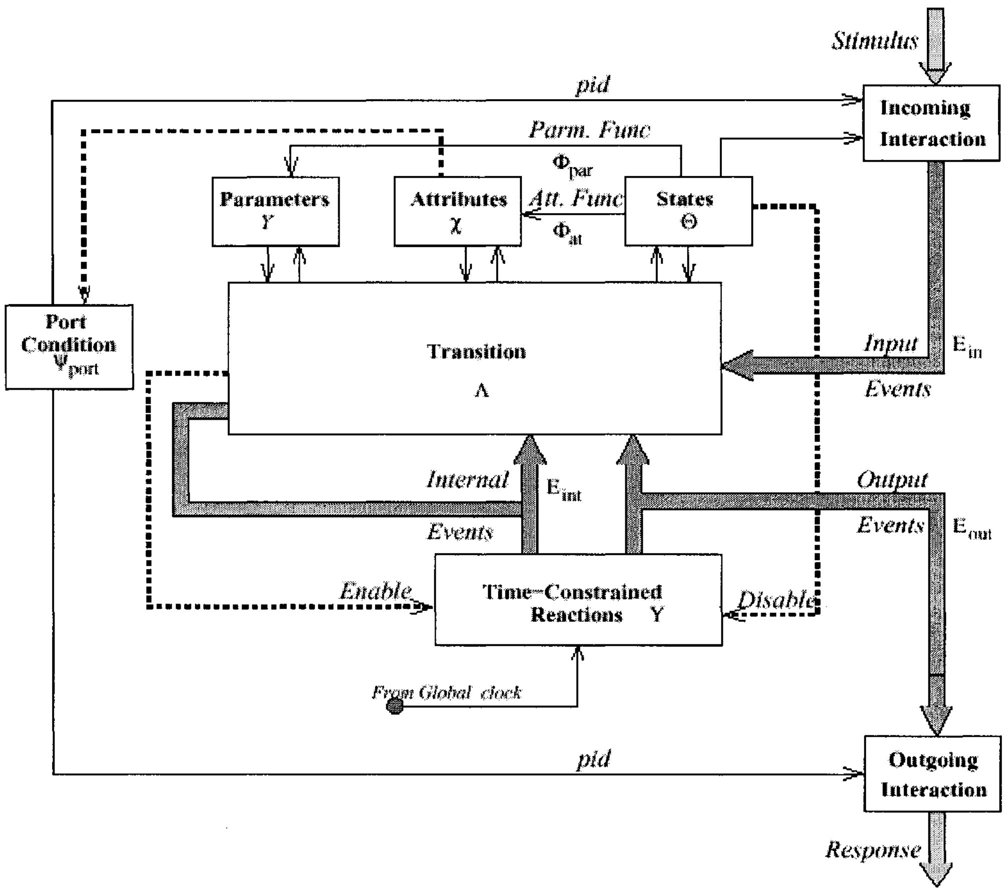


Figure 14: New TROM class anatomy

5.1.2 Semantics of Parameterized Events

TROM formalism contains two kinds of parameters: *constants* and *variables*. A constant parameter may be a constant value of the following types: *integer*, *char*, or *string*. An object in the system may not change the constant parameter's value. In contrast, , the value of a *variable* may be changed at running time. There two kinds of variables: *static* and *dynamic*. A "$" symbol precedes a *static variable,* which has a global scope. In contrast, the scope of a *dynamic variable* is the TROM object that defines it. Since TROM formalism does not support global declarations, parameters have to be declared in all classes that will use them.

Constant and *variable* parameters behave similarly to attributes, with some subtle differences. First, parameters differ in the way that they associate with states of an object. For each state, the parameter function defines a subset of parameters that are active in the state. Parameters that will be associated with a state are identified from incoming and outgoing transitions of the state. The parameters that appear in the post-condition of the incoming transitions are associated with the state. However, the parameters that appear onthe parameter list of the outgoing transitions are also associated with the state. Second, the usage of parameters throughout the behavior of the TROM is different from the behavior of the attributes. A parameter list may appear in any transition and may update the values of the parameters to affect the enabling condition and post-condition of the transition.

5.2 Extensions to the Parameterized Events

Haydar's extension to parameterized events [Hay01] limited parameters to one of the following primitive types*:* *nat*, *integer*, *string*, or *boolean*. These primitive types are not sufficient for a complex system to exchange data. Fox example, in the robotics case (see Chapter 7), the type of parameter is trait *Part* and the assignment to the parameter will be obtained from a function. Therefore, both the type of the parameter and the value of the expression need to be extended as follows:

- First, the parameter must be able to be of trait type; and
- Second, the value must be able to be obtained not only from a mathematical expression but also from the return value of a function.

In addition, because Haydar [Hay01] did not define the syntax of the simulation event list, we have to extend its syntax to accommodate parameterized events.

5.2.1 Extension to the Type of Parameter

Because the syntax of the parameter expression is changed, the syntax of the transition specification needs to be redefined as follows:

Trans_specs	::=	Transition-Specifications: NL <tran_spec_list>
tran_spec_list	::=	<tran_spec> | <tran_spec> <tran_spec_list>
tran_spec	::=	<tran_spec_name>: <state_pairs> <trig_event> <assertion>→<assertion>; NL
state_pairs	::=	<state_pair>; | <state_pair>; <state_pairs>;
state_pair	::=	(<state_name>, <state_name>)
trig_event	::=	<event_name> <parm_list> (<assertion>)
parm_list	::=	NIL | [<parm_expression > <more_parm_entries>]
parm_expression	::=	<parm_name> | <parm_name> = <value>
more_parm_entries		NIL | , <parm_expression> <more_parm_expression>
parm_name	::=	String
value	::=	Nat | Integer | String | Boolean | Trait

assertion	::=	<simple_exp> | <simple_exp> <b_op> <simple_exp>
b_op	::=	= | ≠ | > | ≥ | ≤ | <
simple_exp	::=	<term> | <term> <OR> <term>
term	::=	<factor> | <factor> <AND> <factor>
factor	::=	<NOT> <factor> | pid | <att_name'> | <att_name> | true | false | <LSL_term> | (<assertion>)
LSL_term	::=	<LSL_func_name> (<arg_list>)
arg_list	::=	<arg> | <arg>, <arg_list>
arg	::=	Pid | <att_name> | <LSL_term>
att_name'	::=	String
att_name	::=	String
state_name	::=	String
event_name	::=	String
LSL_func_name	::=	String
OR	::=	|
AND	::=	&
NOT	::=	!

Table 1: Modified grammar for transition specification

5.2.2 Extension to the Syntax of Simulation Event List

The Simulation Event List (SEL) defines sample environmental events of the system forsimulation. As a part of the extension to parameterized events, its grammar also should be redefined.

SEL_spec	::=	SEL:<subsystemname> NL {<eventlist>}^{n} end
subsystemname	::=	String
eventlist	::=	<event>, <trom_obj_name>, < port_type_name>, < occur_time >; NL
event	::=	<event_name>[<valuelist>]
event_name	::=	String
valuelist	::=	NIL | <parm_expression> | < parm_expression >, <valuelist>
parm_expression	::=	<parm_name> = <value>
parm_name	::=	String
value	::=	Nat | String | Integer | Boolean | Trait
trom_obj_name	::=	String
port_type_name	::=	@String
occur_time	::=	Integer

Table 2: Modified grammar for SEL specification

5.3 Upgraded Interpreter

To accommodate the extended new grammar, the Interpreter needed to be updated. The following presents the previous design [Hay01] and shows the new design.

5.3.1 The Previous Design with Parameterized Events

Haydar's design [Hay01] changed the following parameters to reflect the new formalism:

- *Abstract Syntax Tree* structure: only *TROMclass* and *event* class were changed. One class–*Trans_parmlist*–was added into class *TROMclass* to store parameters and their possible values in each transition, and another class – *Parmlist*– was added into class *event* to store all parameters that can be carried by the event in a TROM class.
- TROM *class parser*: the TROM class parser was changed to enable the parsing of the *Parameter-Specifications* section and argument list of the triggering event in each transition. It allows the appearance of simple mathematical expressions in the parameter expressions.
- *Semantic Checks*: It checks that all parameters in the *Parameter-Specifications* section and the argument list of triggering event of transitions are defined in the *Attributes* section.

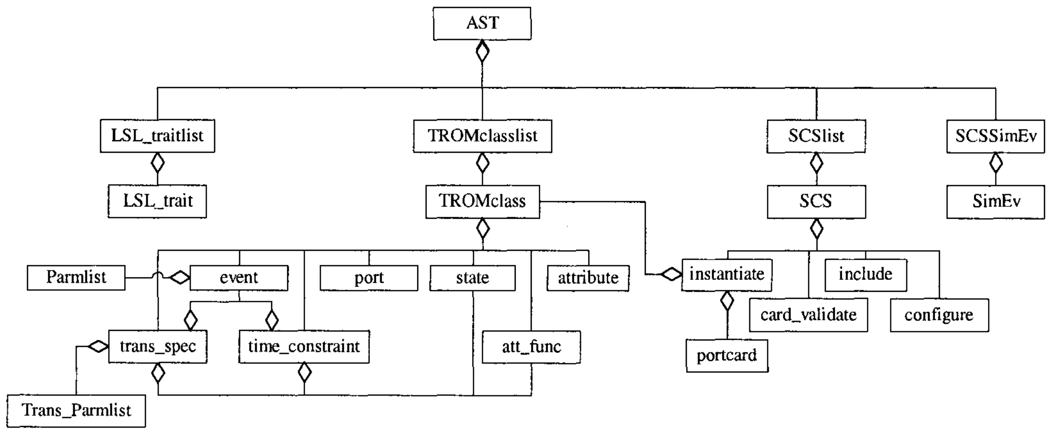


Figure 15: Interpreter Class Diagram (old)

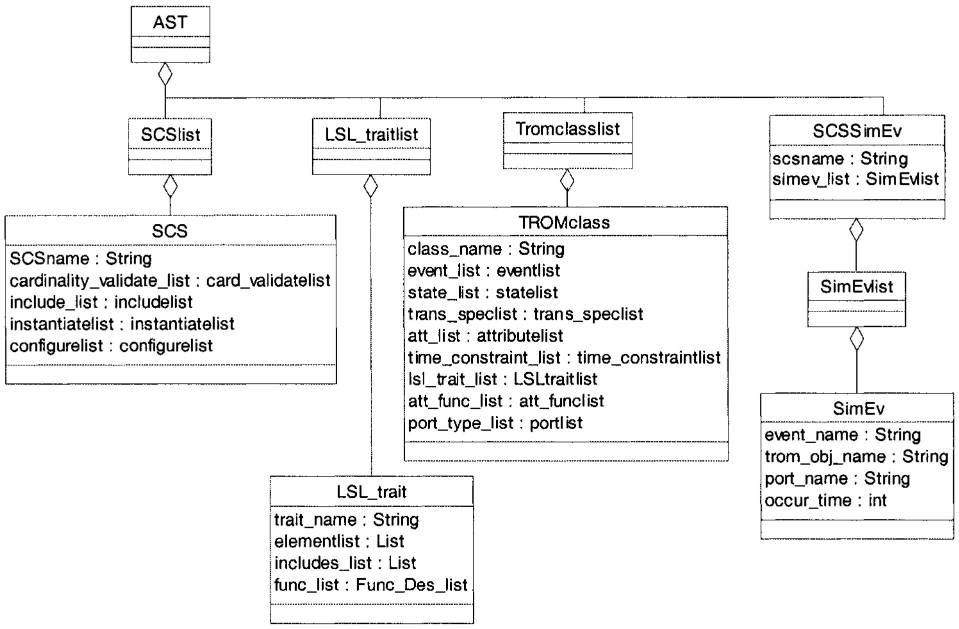


Figure 16: Interpreter Class Diagram – Detailed (old)

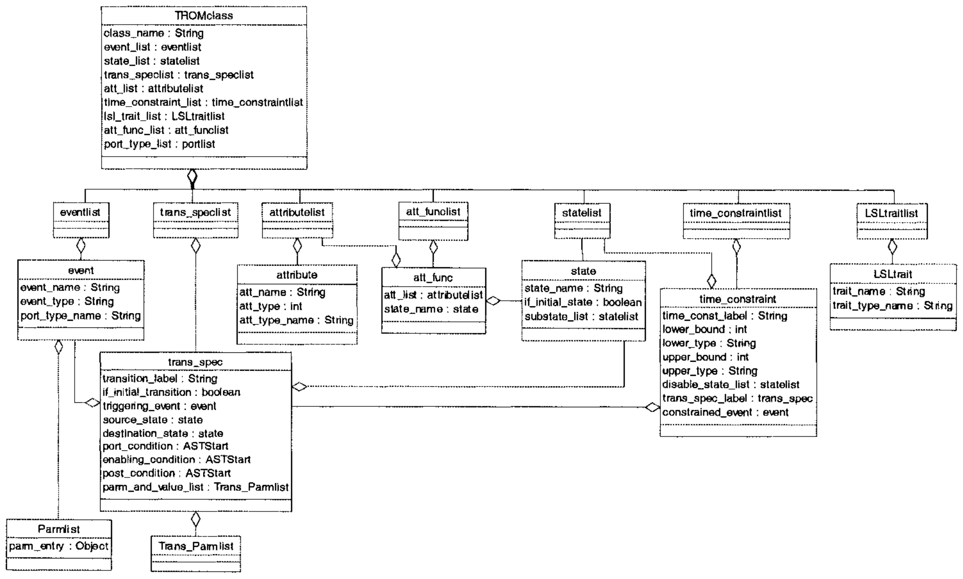


Figure 17: Interpreter Class Diagram – TROMclass (old)

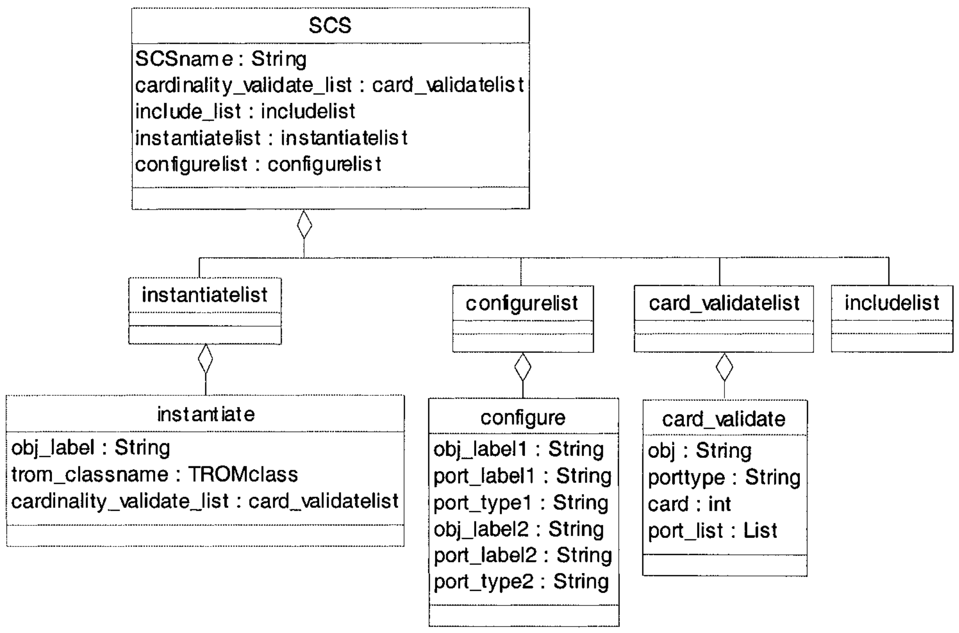


Figure 18: Interpreter Class Diagram – SCS (old)

5.3.2 Upgraded Interpreter

In the new design, some classes need to be added and some components need to be upgraded. JJTree is the preprocessor of JavaCC for parsing complex expressions and building a parse tree. Class *ASTStart* is a production of JJTree for representing complex expressions. The JJTree parser is embedded into JavaCC to parse the parameter expressions. The following discusses the changes.

- Class *ParmValueList*: A new class *ParmValueList* was added into the Interpreter module to store the parameter expressions. Its elements are of type class *ASTStart,* which is a syntax tree that represents the complex parameter expression. For example, the value of the parameter may be a Boolean expression consisting of functions.
- *Abstract Syntax Tree* structure: class *SimEV* was changed to reflect the new grammar in the Simulation Event List (SEL). Class *Trans_Parmlist* was changed to enable storage of complex parameter expressions.
    - To compile complex parameter expressions, class *ParmValueList* replaced the parameter list in the class *TROMclass* of type class *Trans_Parmlist.*

- A parameter list, of type class *ParmValueList*, was added into class *SimEv* to store the parameter expression in the argument list from theSimulation Event List's environmental event.

- JJTree *parser*: the JJTree parser was upgraded to parse port-, pre-, and post-conditions, as well as complex parameter expressions.
- TROM *class parser*: the TROM class parser was changed to parse complex parameter expressions in the argument list of the triggering event. Each parameter expression is parsed into a class *ASTStart* and added into the parameter list, of type class *ParmValueList.*
- SEL *parser*: the parser of SEL was upgraded to parse the argument list of the environmental event. Each parameter expression is parsed into a class *ASTStart* and added into the parameter list, of type class *ParmValueList.*
- LSL Trait *parser*: According to the TROM formalism, the name space of the functions in the LSL Trait is divided into two spaces: one is in the local namespace of a trait; the other is in the global namespace. Functions with at least one argument are in the distinct local namespace of a trait because the arguments are a part of a function's signature for distinguishing from other functions. Functions without any arguments are in the global namespace. The class *ConstTable* was created in the LSL Trait parser to record the functions without arguments, and their function names must be unique in the class *ConstTable*. The class *ConstTable* only has a single instance and an attribute of *ConstTableMap* class type for storing the functions.
- *Semantic Checks*: new semantics were added after the complex parameter expression was introduced and SEL parser was upgraded.
  - All parameters in the *Parameter-Specifications* section and the argument list of triggering event of transitions should be defined in the *Attributes* section.
  - In SEL, the number and order of parameters in the argument list of the environmental event should be equal to that of the same event in the corresponding transition.
  - The value should be true or false if the data type of the parameter is Boolean, and the value should be digital if the data type of the parameter is Integer.
  - If the data type of the parameter is Trait, the value will be obtained from the return value of a function. Then, the interperter will check that the return type

of the function is identical to the type of parameter. In addition, it will check that the function name appears in the corresponding trait definition, and that the number of its arguments should be the same.

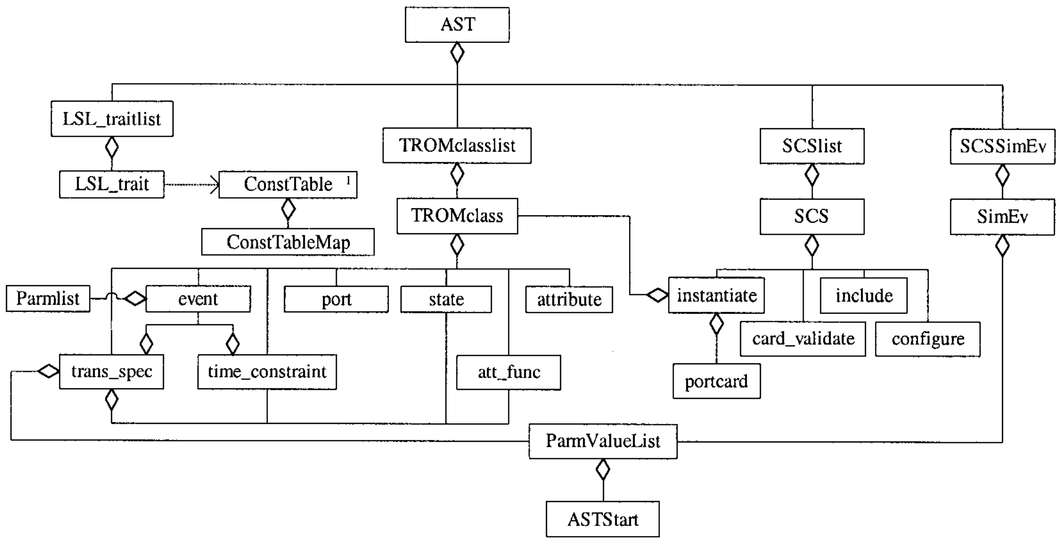


Figure 19: Interpreter Class Diagram (new)

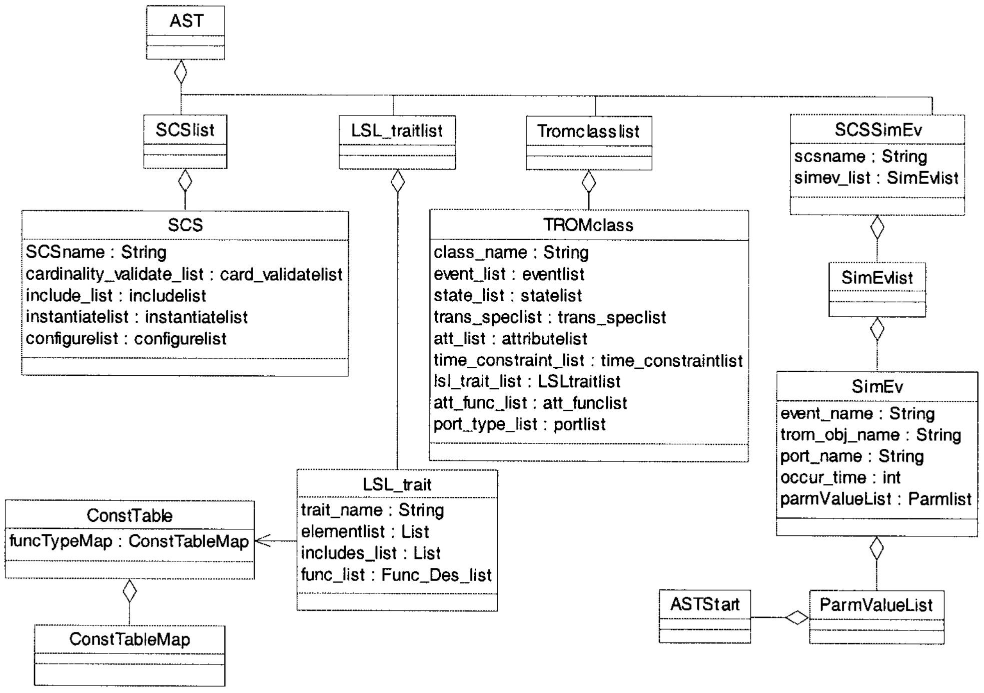


Figure 20: Interpreter Class Diagram – Detailed (new)

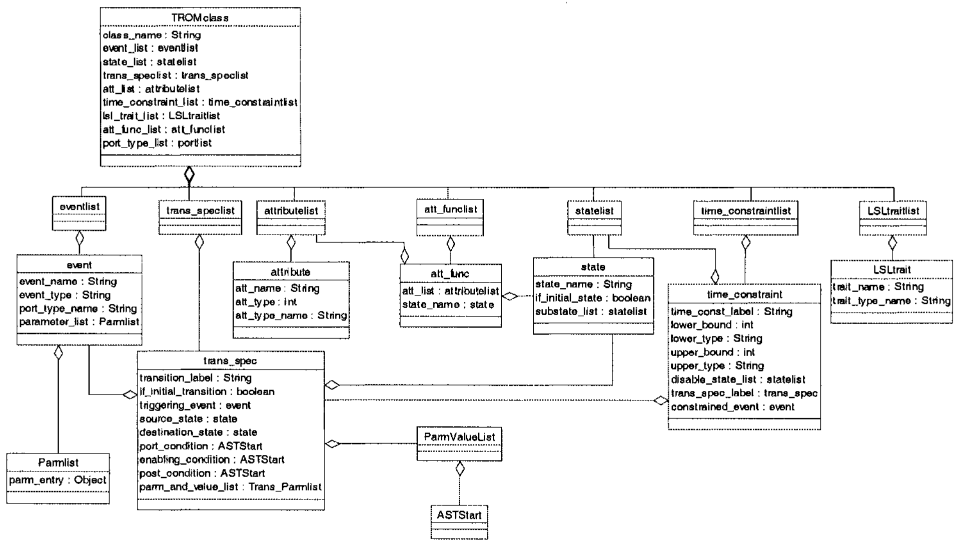


Figure 21: Interpreter Class Diagram – TROMclass (new)

5.4 Upgraded Simulator

After parameterized events were introduced into the TROM formalism, not only were some new classes added, but the simulation algorithm also had to be changed. The following subsections discuss the simulator's previous design and show the new design and updated simulation algorithm.

5.4.1 Previous Design

After the interpreter compiled the TROM specifications, it generated an internal representation *Abstract Syntax Tree* (AST), which was inputted to the simulator. The AST recorded all static information about the TROM specifications. The simulator used the AST to create a set of classes headed by class *Subsystem*, which represented dynamic information while performing the simulation. The class *Subsystem* records the runtime information about a whole system; in contrast, the class TROM records the runtime information about a TROM object in the subsystems.

Figure 23 shows the structure of class *Susbystem.* It consists of a set of included subsystems, a set of TROM classes, and a set of *port links* that represent the configuration between the TROM objects. The class *SubsystemObjectSupport* is used to initialize all subsystems and their included components at the beginning of the simulation.

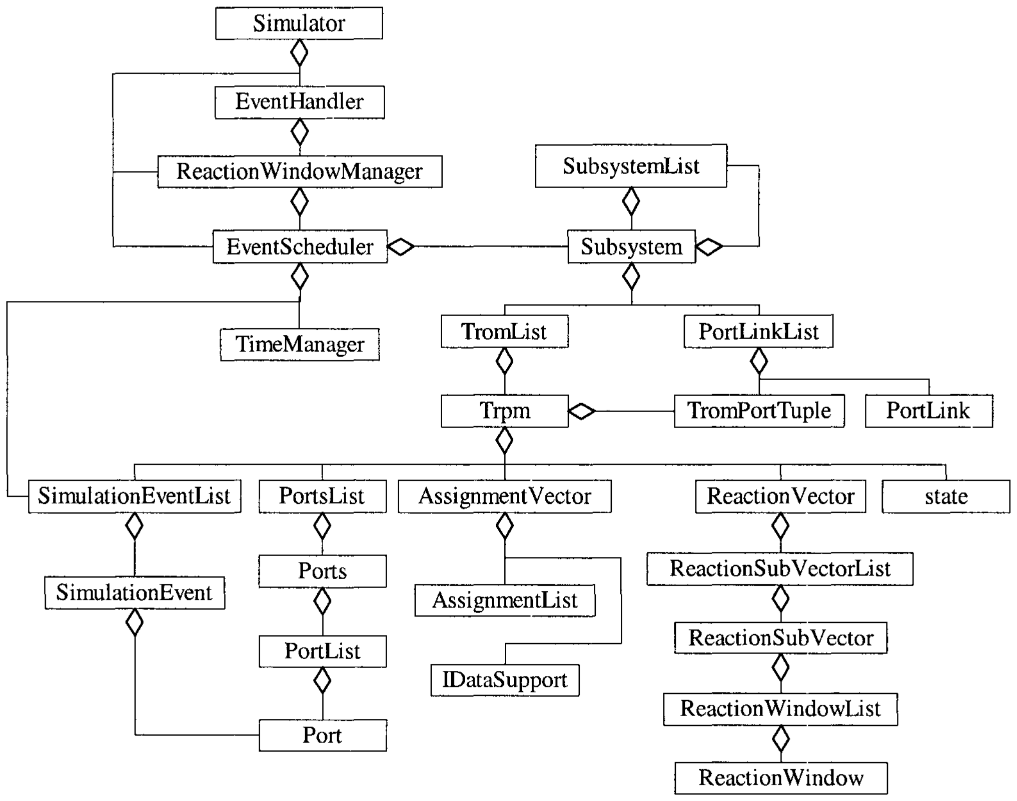


Figure 22: Simulator Class Diagram (old)

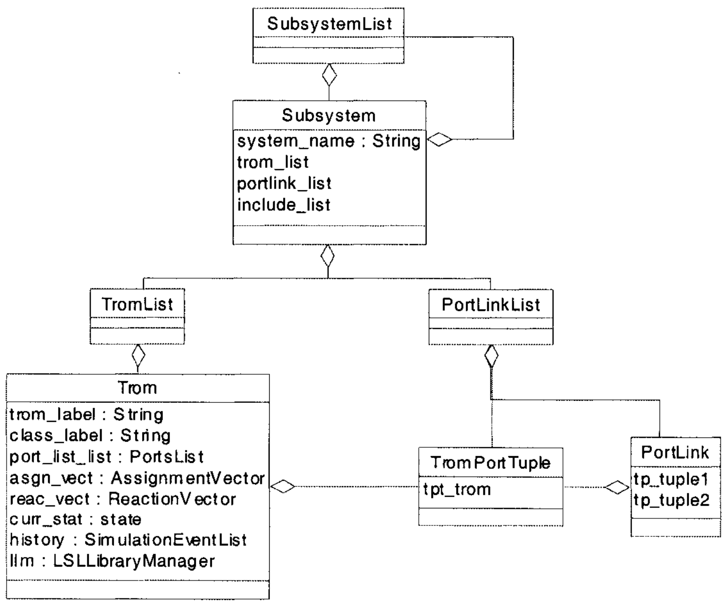


Figure 23: Simulator Class Diagram – Subsystem (old)

The TROM class comprises of a port link, current state, assignment vector, and reaction vector. All these classes together record the dynamic data of a TROM object.

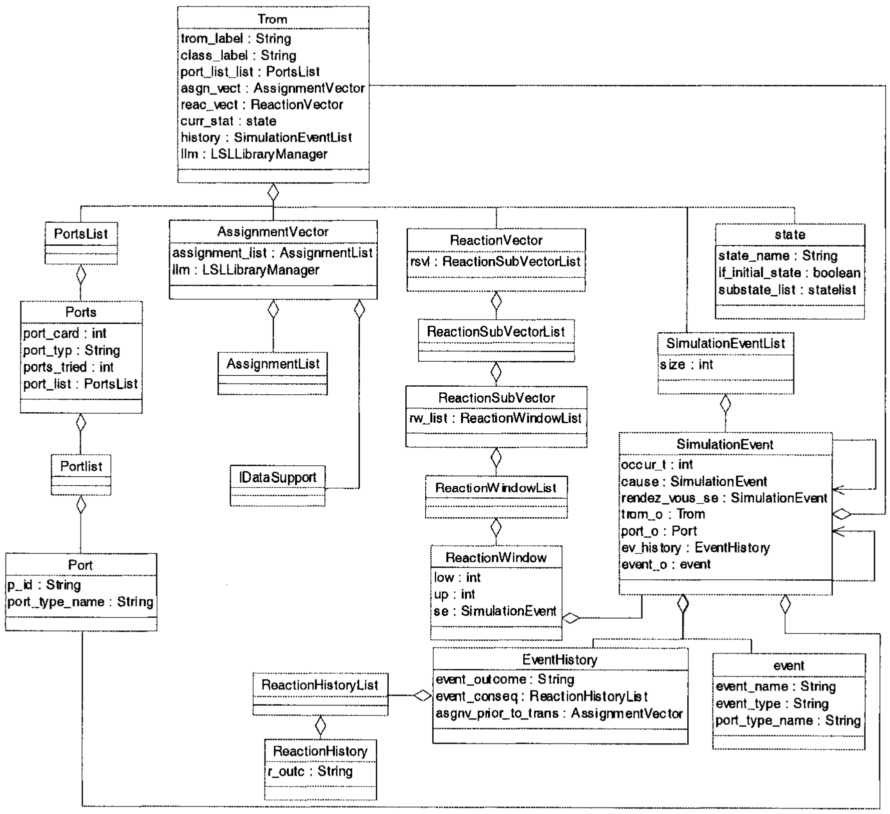


Figure 24: Simulator Class Diagram – TROM (old)

The class *SimulationEvent* represents a computational step of the simulation.

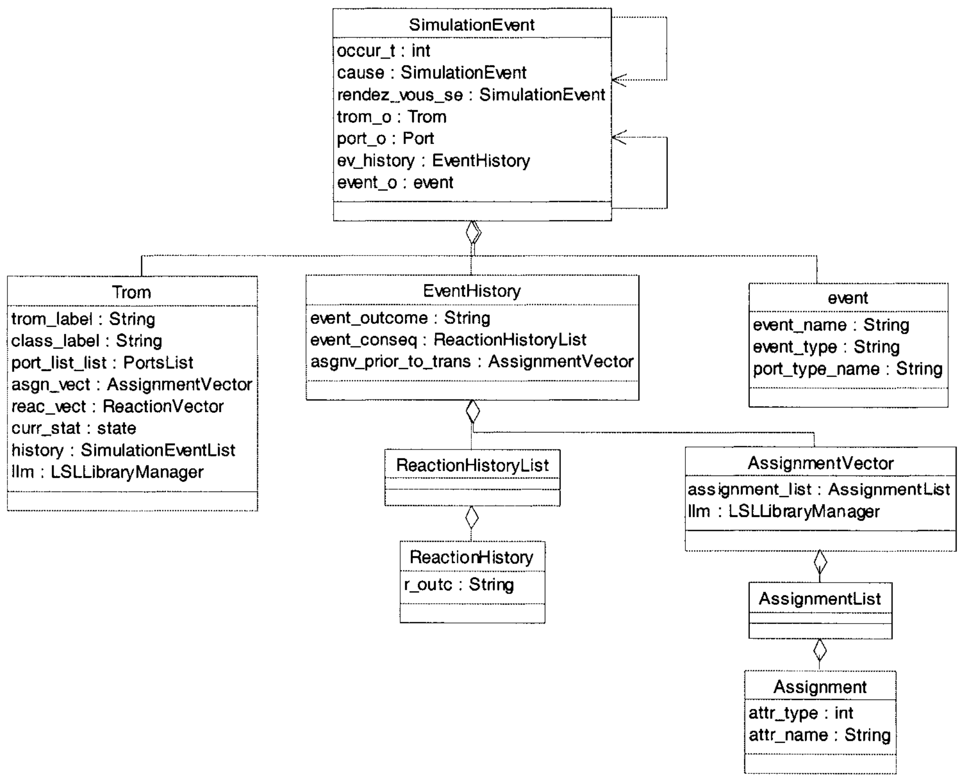


Figure 25: Simulator Class Diagram – SimulationEvent (old)

5.4.2 New Design

Once parameterized events were introduced, classes were needed to handle the new information. In the original design without parameters, the TROM objects did not share attribute values. In the new design with parameterized events, all TROM class may view the *static* parameters. The *dynamic* parameters and the values they carry need to be passed between TROM objects. We also need to change the simulation algorithm to enable the swapping of the *dynamic* parameters' values. The following discusses them from two aspects.

- To simulate system modeling with parameterized events, some new classes were added to the simulator to work with the new algorithm.
  - A *parameter update list* was added into the class *SimulationEvent* to store a list of parameter expressions for updating parameters. These parameter expressions may come from the argument list of the environmental events in the *Simulation Event List* (SEL) or the argument list of the triggering events in the transition specifications.

- The scope of *dynamic* parameters is in the TROM object that defines the parameters. When a parameterized output event of a TROM object is issued, it will transfer the *dynamic* parameters and their values to the corresponding parameterized input event of another TROM object. To finish the transportation of parameters between TROM objects, class *TempVarSwapList* was introduced into the simulator to swap parameters. It only has a single instance and its only attribute is of class *AssignmentList*, which is used to store swapped parameters.
- For enabling defining static parameter, a new class *StaticVarList* is introduced into the Simulator. The class *StaticVarList* only has a single instance and its only attribute is of type of class *AssignmentList* used for storing global variable. The parameters in the class *StaticVarList* are shared by all TROM objects in the system.

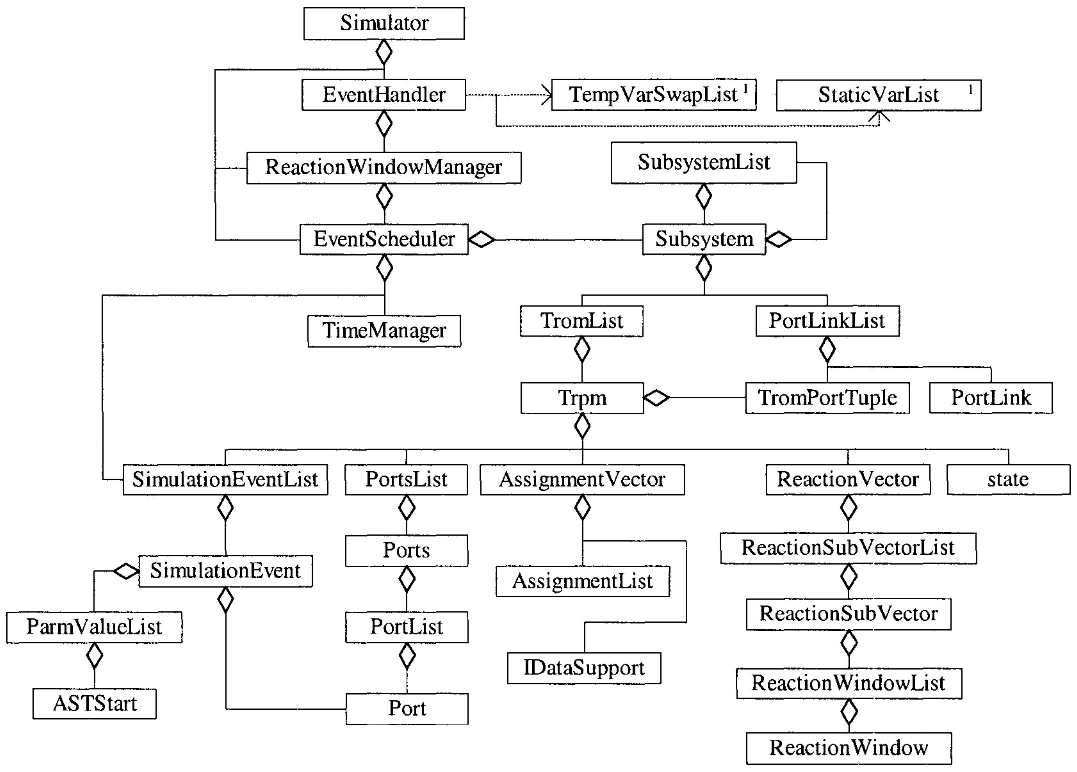


Figure 26: Simulator Class Diagram – SimulationEvent (new)

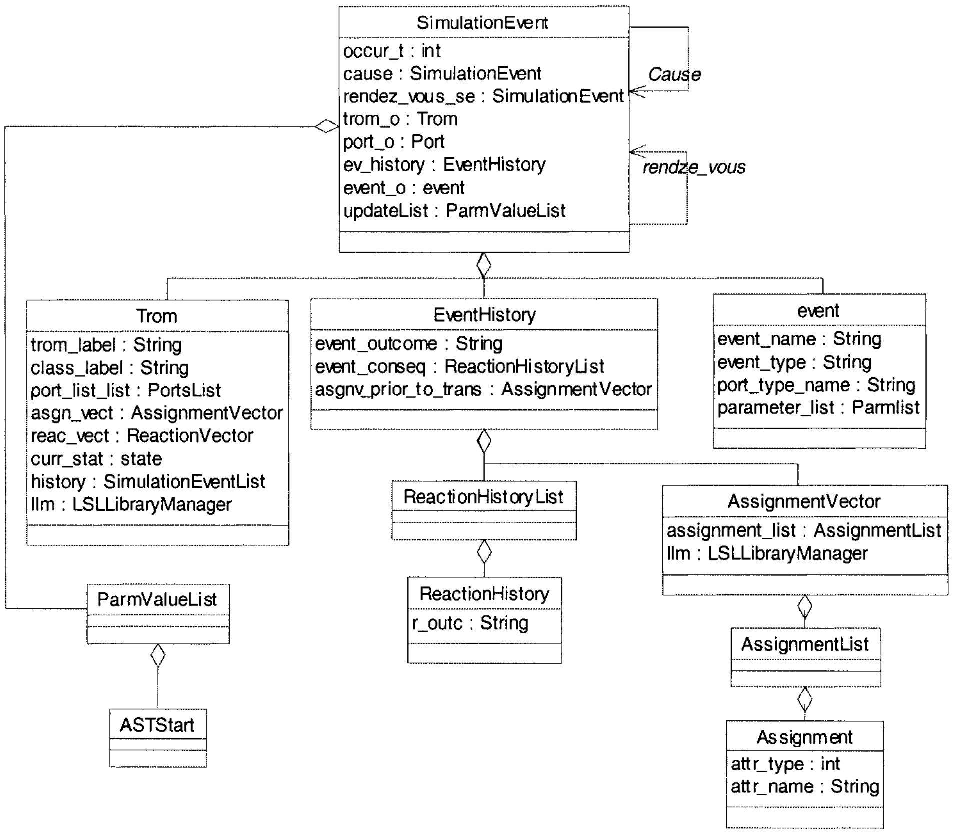


Figure 27: Simulator Class Diagram – SimulationEvent (new)

- The following describes the parameter-related algorithm, including parameter swap between TROM objects:
  - The parameter values of the internal event from the initial state are null
  - When a transition occurs, other events could be scheduled if some time constraints indicate that the transition will enable these events.
  - Each event has a *parameter update list* that includes some assignment expressions that will update the parameters' values. These parameter expressions may come from the argument list of the environmental events in Simulation Event List or from the argument list of the triggering event in the transition specification.
  - Output event handling: the output event will assign parameters and their values to the Temporary Variable Swap List if the parameters are dynamic.
    1) Get transition corresponding to this event.
    2) Check transition and update the parameter update list with parameter expressions in the argument list of the triggering event.

3) Update the parameters in TROM objects or *StaticVarList* by calculating the parameter expressions in the parameter update list.
4) Evaluate post-condition.
5) Clone all *dynamic* parameters, which appear in the argument list of the triggering event, with their values and insert them to the Temporary Variable Swap List.
6) Schedule all events constrained by this event.
7) Schedule corresponding input events.

- Input event handling: the input event will read parameters and their values from the Temporary Variable Swap List and clear the list after that.
    1) Get parameter values from the Temporary Variable Swap List and update the parameters.
    2) Clear the Temporary Variable Swap List
    3) Get the transition corresponding to this event
    4) Evaluate the post-condition
    5) Schedule all events constrained by this event

The next section shows the new simulation algorithm.

5.4.3 Simulation Algorithm

```
begin /* simulation algorithm */
    type-check TROM class and subsystem specifications
    preprocess TROM classes to be used in simulation
    get label of Subsystem s to simulate
    instantiate Subsystem s
    instantiate TROM objects for each Subsystem
    create PortList for each PortType for each TROM object according to port cardinality
    initialize current state and assignment vectore of each TROM object
    configure PortLinks for each Subsystem
    initialize SimulationClock
    for each object trom of a subsystem
    begin /* sort constrained-output events of an GRC to make them issued orderly based on the time
                interval of their TimeConstraint */
        get TimeConstraintList tcl associated with the TROM object trom
        for all TimeConstraint in TimeConstraintList tcl
        begin
            get first TimeConstrait tc from TimeConstraintList tcl and insert it into a temporary
                    TimeConstraintList tl
            remove TimeConstrait tc from TimeConstraintList tcl
            for all remaining TimeConstraints in TimeConstraintList tcl
            begin
                if TimeConstraint rtc has same transition label as the TimeConstraint tc in temporary
                        TimeConstraintList tl
                begin /* insert TimeConstraint rtc into temporary TimeConstraintList tl by
                        comparing their time interval */
                for all TimeConstraints in temporary TimeConstraintList *tl*
                begin
                    get TimeConstraint *tc* from temporary TimeConstraintList *tl*
                    if the *upper bound* of TimeConstraint *rtc* is less than that of TimeConstraint
                                *tc*
                    begin
                        insert TimeConstraint *rtc* at the beginning of the temporary
                                TimeConstraintList *tl*
                    end
                    else if *lower bound* of TimeConstraint *rtc* is less than that of TimeConstraint
                                *tc* and their *upper bounds* are equal
                    begin
                        insert TimeConstraint *rtc* immediate before TimeConstraint *tc*
                    end
                    else add TimeConstraint *rtc* at the end of the temporary TimeConstraintList
                                *tl*
                    end
                end
                remove TimeConstraint rtc from TimeConstraintList *tcl*
            end
            insert all TimeConstraint *tc* in temporary TimeConstraintList *tl* into a new
                    TimeConstraintList *ntl*
        end
    end
    schedule *all output-and-unconstrained* events and put the parameter expressions getting from
            the initial Simulation Event List into the UpdateValueList *parmList*
    schedule *unconstrained internal* events from *initial* state for each TROM object with a *null*
            UpdateValueList *parmList*
    for all SimulationEvent *se* in SimulationEventList *sel*
    begin /* *at this stage SimulationClock can be frozen and debugger activated* */
        While *SimulationClock* < *occur time* of SimulationEvent *se*
        begin
            increment *SimulationClock* /* *using machine clock* */
        end
        While exists SimulationEvent *se* and *SimulationClock==occur time* of *se*
        begin /* *handle SimulationEvent se* */
            get TROM object *trom* accetping SimulationEvent se from Subsystem *s*
                    /* *get parameter values from the temporary value swap list* */
            if event is an input event
            begin
                for all parameters in the TempVarSwapList *tvsl*
                begin
                    get parameter parm with its value v
                    get parameter parm from the AssignmentList al in the Trom trom
                    update the parameter with value v
                end
            end
            get TransitionSpec ts triggered by SimulationEvent se
                    /* *update parameter with assignment expression in the UpdateValueList* */
            if exist any assignment expressions SimpleNode *n* in the ParmValueList
                        *transParmValueList* of transition specification
            begin
                for all assignment expressions SimpleNode *n* exist in the transition specification
                begin
                    get SimpleNode *n* from the ParmValueList *transParmValueList* in the
                                transition specification
                    insert SimpleNode n into the UpdateValueList selParmValueList
                end
                for all assignment expressions SimpleNode n UpdateValueList
                            *selParmValueList*
        begin
            update parameters in the AssignmentList *al* through calculating SimpleNode
                                    *n*
        end
        if event is an output event
        begin
            for all parameters *p* in the Parmlist *pl* of the event
            begin
                if parameter *p* is not static variable
                begin
                    obtain parameter value *pv* from the AssignmentList *al* in the Trom
                                        *trom*
                    clone a new value *npv* for *pv*
                    add the new value *npv* to the TempVarSwapList *tvsl*
                end
            end
        end
    end
    /* update history of SimulationEvent se */
    save current state of TROM object trom in EventHistory of se
    save assignment vector of TROM object trom in EventHistory of se
                                /* update status of TROM object trom */
    change current state of TROM object trom to destination of TransitionSpec ts or to
    entry state of destination state of TransitionSpec ts if a complex state
    change assignment vector of TROM object trom accordeing to postcondition of ts
    /* handle transition specified by TransitionSpec ts */
    for all TimeConstraint tc in list of TimeConstraints for TROM object trom
    begin
        if *constrained event* of TimeConstraint *tc==label* of SimulationEvent *se*
        begin
            for each ReactionWindow rw in reactoin subvector associated with tc
            begin
                if SimulationEvent se occurs within ReactionWindow rw
                begin /* fire reaction according to TimeConstraint tc */
                    remove ReactionWindow *rw* from *reaction subvector* associated
                                with *tc*
                    insert ReactionHistory *rh* in EventHistory of *se* according to *rw*
                end
            end
        end
        if *current* state of TROM object *trom* is in set of *disabling* states of *tc*
        begin /* *disable reaction according to TimeConstraint tc* */
            for all ReactionWindows *rw* in reaction *subvector* associated with *tc*
            begin
                remove ReactionWindow *rw* from reaction *subvector* associated with *tc*
                insert ReactionHistory rh in EventHistory of se according to rw
                unschedule disabled SimulationEvent in SimulationEventList sel
                if constrained event of TimeConstraint tc is an output event
                begin
                    remove disabled SimulationEvent scheduled for synchronization
                end
            end
        end
        if label of TransitionSpec ts == transition label of TimeConstraint tc
        begin /* enable reaction according to TimeConstraint */
            insert new ReactionWindow rw in reaction subvector associated with tc
                    based on the InternalClock baseTime of TROM object trom
            insert ReactionHistory rh in EventHistory of se according to rw
            insert new SimulationEvent se2 in SimulationEventList sel using LRU port
                    of PortType of constrained event of tc and random time within
```

```
                        ReactionWindow rw and parameter-value map of SimulationEvent
                        se
                    set the InternalClock baseTime of TROM object trom as the due running
                        time of the new SimulationEvent se2
                end
            end
            schedule unconstrained internal event from current state for TROM object trom and
                    reset all values in parameter value list to null
            if constrained event of TimeConstraint tc is an output event
            begin /* identify linked TROM object for synchronization */
                get PortLink pl from Subsystem s linking the two TROM objects
                get parameter-value map pvm from the output SimulationEvent se
                insert new SimulationEvent se3 in SimulationEventList sel, using PortLink pl
                        and parameter-value map pvm, for synchronization
            end
            get next SimulationEvent se from SimulationEventList sel
        end
    end
  end
end
```

After the parameterized events were implemented in the validation tool, we used it to analyze the Train-Gate-Controller and Robotics case studies with parameterized events. The original case studies, their remodeling with parameterized events and the results from the simulation are described in the next two chapters.

Chapter 6

Train-Gate-Controller Case Study

This chapter introduces the Train-Gate-Controller (TGC) problem, a benchmark case study adopted by the real-time research community.

6.1 Problem Description

In our TGC model, three trains pass through two gates by coordinating with two controllers. The trains communicate with the controllers by sending messages, and the controllers control the gates by issuing commands. When a train approaches a gate, it sends a message to the controller of that gate. The controller, in turn, instructs the gate to close. After the train has passed through the gate, it sends another message to the controller, who then commands the gate to open.

The following time constraints must be obeyed:

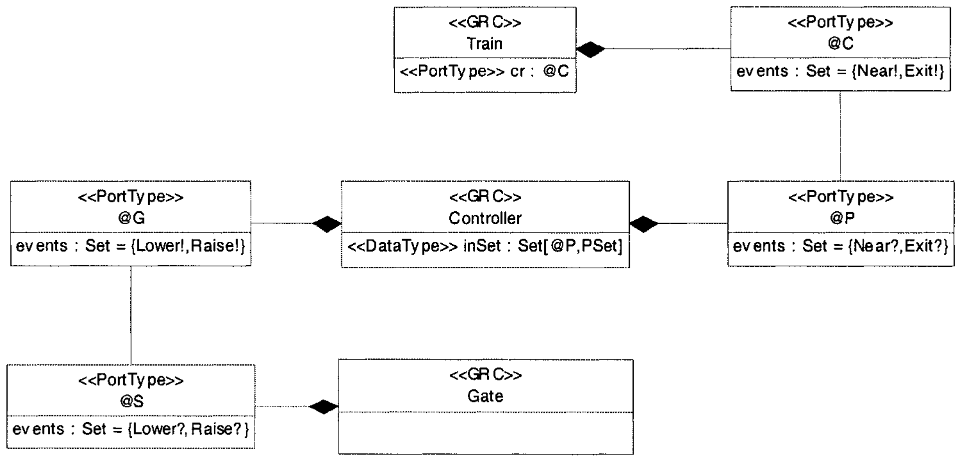
- A train should enter the gate between 2 to 4 time units after it had sent an initial message informing the controller of its arrival.
- A train should leave the gate within 6 time units after it had sent an initial message informing the controller of its arrival.
- The controller should instruct the gate to lower its arm within 1 time unit after receiving a signal from the train that it will be arriving.
- The controller should instruct the gate to raise its arm within 1 time unit after receiving a signal from the train that it is leaving.
- The gate must lower its arm within 1 time unit after receiving the lowering message from the controller.
- The Gate must raise its arm within 1 time unit after receiving the raising message from the controller.

The following subsections show the formal specifications of the TGC system for the original version with non-parameterized events (Section 6.2) and the remodeled TGC system with parameterized events (Section 6.3).

6.2 Original TGC System

6.2.1 Class Diagram for TGC System

1) Train TROM class is an aggregate of port types @C.
2) Gate TROM class is an aggregate of port types @S
3) Controller TROM class is an aggregate of port types @P and @G



<<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 28: TGC system class diagram

The link between the port type @C of the train and the port type @P of the controller means that the train uses port @C to communicate with the controller through its port @P.

The link between the port type @G of the controller and the port type @S of the gate means that the controller uses port @G to communicate with the gate through its port @S.

6.2.2 Train Class

The Train is the environmental class in the system, and all its output events cannot be constrained. As the train approaches a gate, it sends a ***Near*** message to the controller of the gate. Once the train leaves the gate, it sends an ***Exit*** message to the controller of the gate. The port attribute *cr* identifies a train, and is set to the port that transmits the events of the train.

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

Figure 29: Train TROM class – Class specification

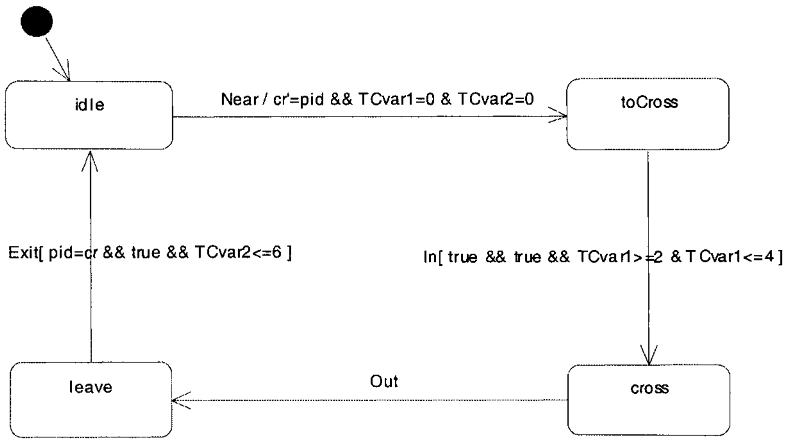

idle
Near / cr'=pid && TCvar1=0 & TCvar2=0
toCross
Exit[pid=cr && true && TCvar2<=6]
In[true && true && TCvar1>=2 &TCvar1<=4]
leave
Out
cross

Figure 30: Train TROM class – State chart diagram

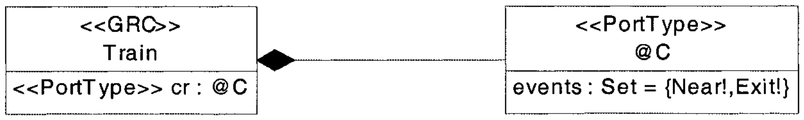


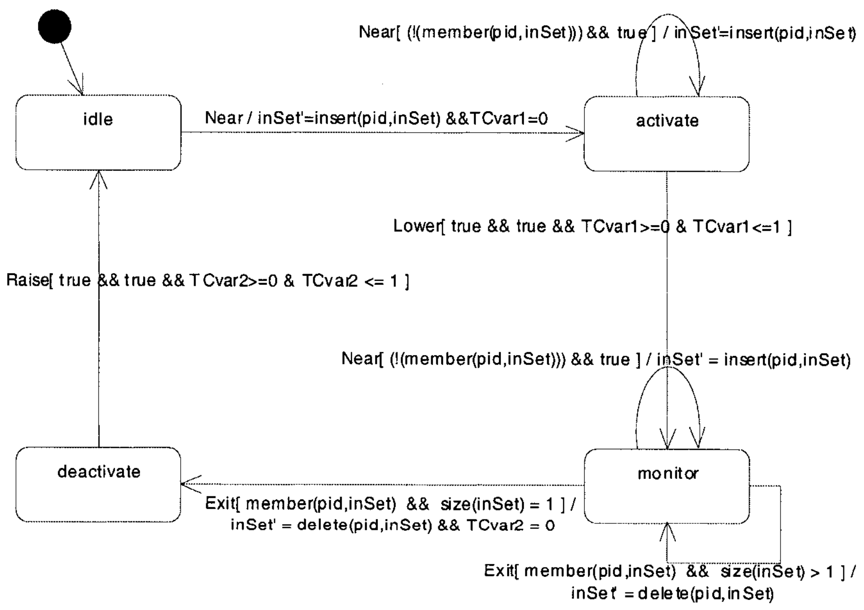
Figure 31: Train TROM class – class diagram

6.2.3 Controller Class

The Controller can communicate with several trains simultaneously and identify them by storing their port value into its attribute ***inSet***. When a trainarrives, it sends a ***Near*** message with its port id ***pid*** to the controller. The controller then stores the ***pid*** into the ***inSet*** and instructs the gate to lower. After the train leaves, the controller will receive an ***Exit*** message from the train and remove the train's ***pid*** from the ***inSet***.

```
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>; Near(!(member(pid,inSet))); true =>
        inSet'=insert(pid,inSet);
     R6: <monitor,monitor>; Exit(member(pid,inSet));
        size(inSet)>1 => inSet'=delete(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 32: Controller TROM class – Class specification



Near[(!(member(pid,inSet))) && true] / inSet'=insert(pid,inSet)
idle
Near / inSet'=insert(pid,inSet) &&TCvar1=0
activate
Lower[true && true && TCvar1>=0 & TCvar1<=1]
Raise[true && true && TCvar2>=0 & TCvar2 <= 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 33: Controller TROM class – State chart diagram

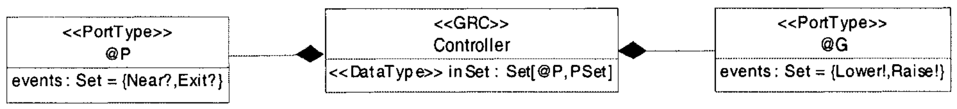

<<PortType>>
@P
events : Set = {Near?,Exit?}
<<GRC>>
Controller
<<DataType>> inSet : Set[@P,PSet]
<<PortType>>
@G
events : Set = {Lower!,Raise!}

Figure 34: Controller TROM class – class diagram

6.2.4 Gate Class

The controller raises or lowers the gate by sending it various messages. The gate lowers after it receives a ***Lower*** message from the controller as the train arrives at the gate, and the gate rises after it receives a ***Raise*** message from from the controller after the train leaves the gate.

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

Figure 35: Gate TROM class – Class specification

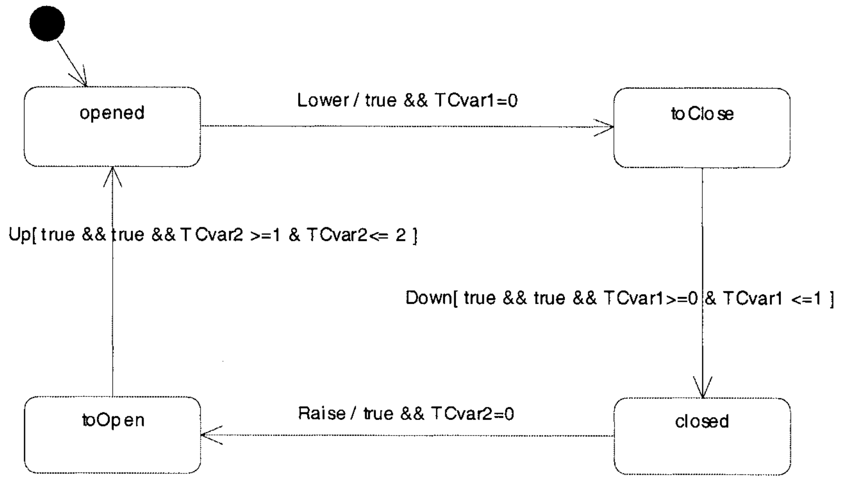

opened
Lower / true && TCvar1=0
toClose
Up[true && true && TCvar2 >=1 & TCvar2<= 2]
Down[true && true && TCvar1>=0 & TCvar1 <=1]
toOpen
Raise / true && TCvar2=0
closed

Figure 36: Gate TROM class – State chart diagram

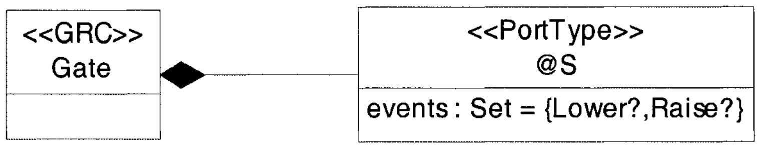


Figure 37: Gate TROM class – class diagram

6.2.5 System Configuration Specification (SCS)

A System Configuration Specification provides the specification for a system or a subsystem by composing reactive classes. A subsystem specification consists of three sections: *Includes*, *Instantiate* and *Configure*. The *Include* section imports other systems. The *Instantiate* section defines objects by parametric substitutions to the cardinality of ports for each port type. The *Configure* section defines the configuration of the system's architecture by composing the specified objects. The composition operator <-> sets up a communication link between compatible ports of interacting objects.

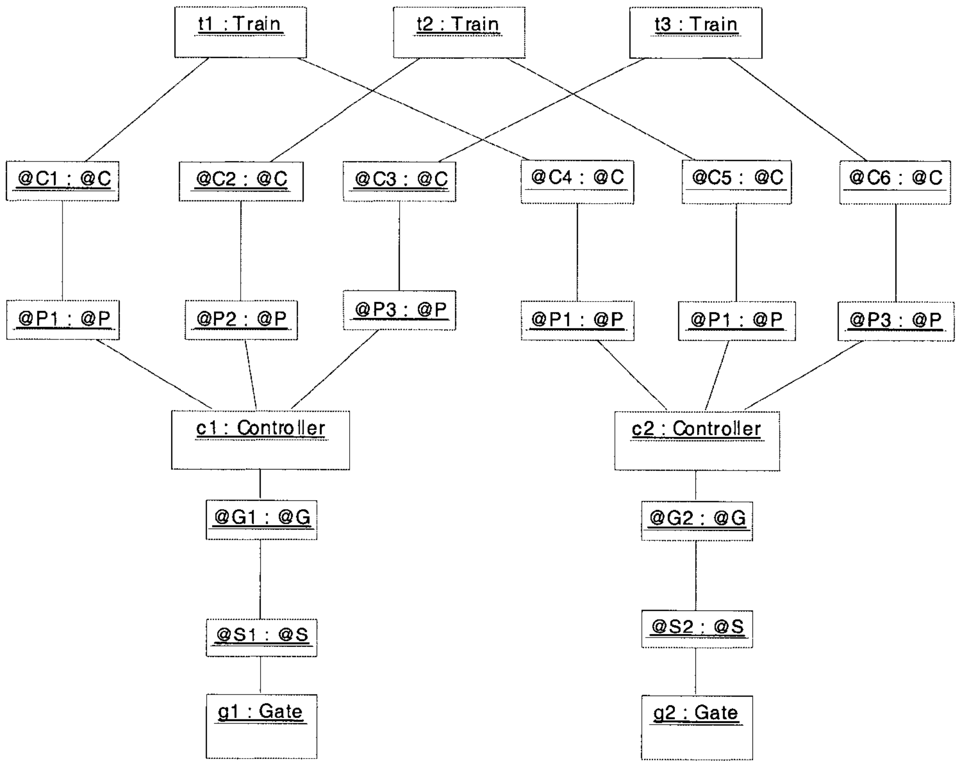


Figure 38: SCS – Collaboration diagram

```
SCS TCG
   Includes:
   Instantiate:
      t1::Train[@C:2];
      t2::Train[@C:2];
      t3::Train[@C:2];
      c1::Controller[@P:3,@G:1];
      c2::Controller[@P:3,@G:1];
      g1::Gate[@S:1];
      g2::Gate[@S:1];
   Configure:
      t1.@C1:@C <-> c1.@P1:@P;
      t1.@C2:@C <-> c2.@P1:@P;
      t2.@C1:@C <-> c1.@P2:@P;
      t2.@C2:@C <-> c2.@P2:@P;
      t3.@C1:@C <-> c1.@P3:@P;
      t3.@C2:@C <-> c2.@P3:@P;
      c1.@G1:@G <-> g1.@S1:@S;
      c2.@G1:@G <-> g2.@S1:@S;
end
```

Figure 39: SCS –Specification

6.2.6 LSL trait

The system uses one trait – ***Set***. The controller uses the trait ***Set*** to store the ports that sent the trains' messages, and to identify the trains.

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

Figure 40: Set LSL trait

6.3 Remodeling the TGC System with Parameterized Events

In the original model of the TGC system, the controller identifies a train by the port used to transmit the *Near* message. After we revised the model, the parameter *TID,* which is of type Integer, now is used to validate the identification of a train. When a train sends an output message to the controller, the parameter *TID* will accompany this message and will be used by the controller to identify the train. The following will show the modified specifications and the simulation results.

6.3.1 Class Diagram for TGC System with Parameterized Events

The class Train defines the parameter TID instead of the attribute port. Similarly, the set *inSet* in the Controller is also changed to store Integer variables.

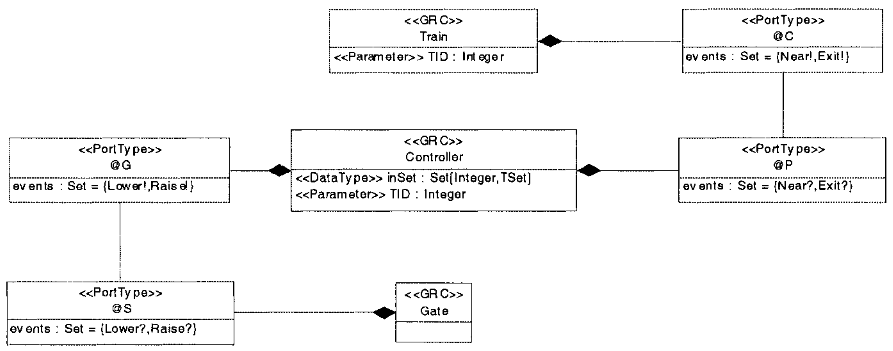

<<GRC>>
Train
<<Parameter>> TID : Integer
<<PortType>>
@C
events : Set = {Near!,Exit!}
<<PortType>>
@G
events : Set = {Lower!,Raise!}
<<GRC>>
Controller
<<DataType>> inSet : Set[Integer,TSet]
<<Parameter>> TID : Integer
<<PortType>>
@P
events : Set = {Near?,Exit?}
<<PortType>>
@S
events : Set = {Lower?,Raise?}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

Figure 41: TGC system class diagram

6.3.2 Train Class with Parameterized Events

The Train TROM class includes a parameter *TID*, which is used to identify the objects which are communicating with the controller.

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

Figure 42: Train TROM class – class specification

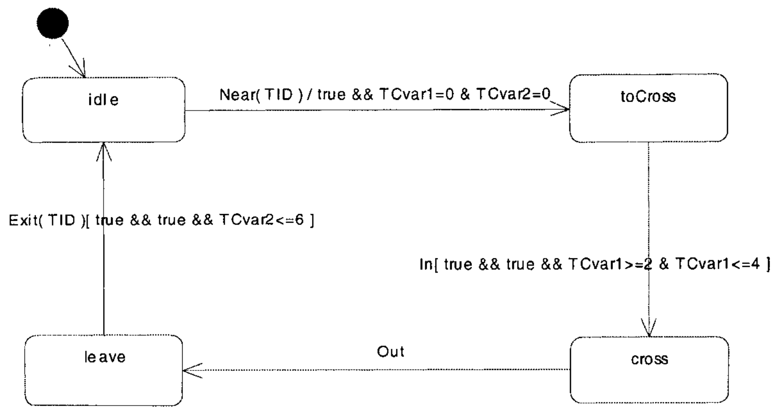

idle
Near(TID) / true && TCvar1=0 & TCvar2=0
toCross
Exit(TID)[true && true && TCvar2<=6]
In[true && true && TCvar1>=2 & TCvar1<=4]
leave
Out
cross

Figure 43: Train TROM class – state chart diagram

6.3.3 Controller Class with Parameterized Events

The controller contains a set *inSet* for storing the *TID*s. It uses these *TID*s to distinguish between different trains and their events.

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

Figure 44: Controller TROM class – class specification

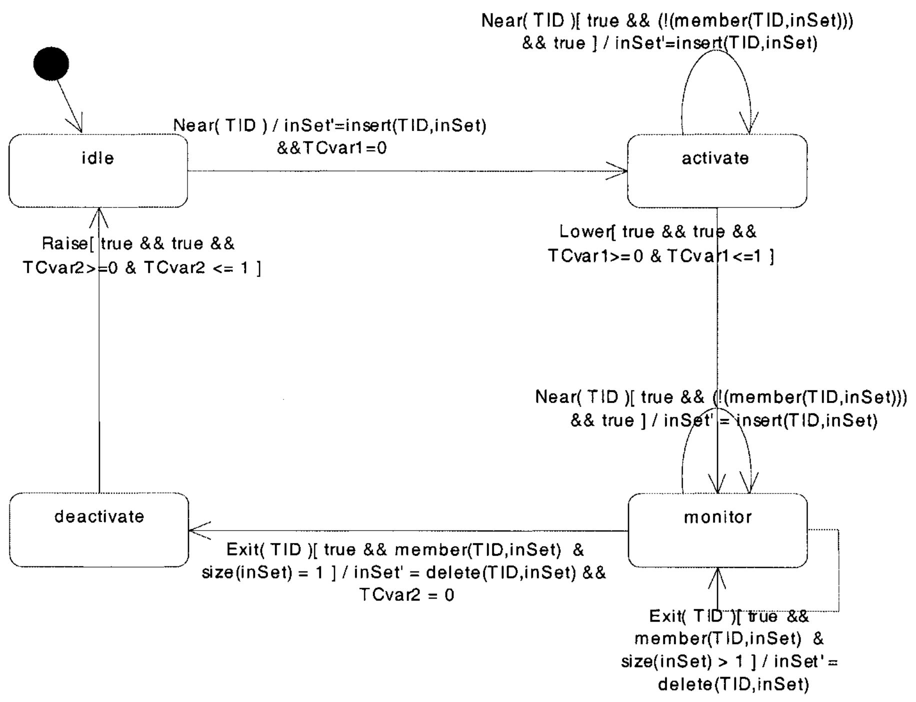

Near(TID)[true && (!(member(TID,inSet))) && true] / inSet'=insert(TID,inSet)
Near(TID) / inSet'=insert(TID,inSet) &&TCvar1=0
idle
activate
Raise[true && true && TCvar2>=0 & TCvar2 <= 1]
Lower[true && true && TCvar1>=0 & TCvar1<=1]
Near(TID)[true && (!(member(TID,inSet))) && true] / inSet' = insert(TID,inSet)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

Figure 45: Controller TROM class – state chart diagram

6.3.4 Gate Class with Parameterized Events

Because the class Gate does not need to know which train is crossing, it does not define the parameter *TID*.

```
Class Gate [@S]
Events: Lower?@S, Down, Up, Raise?@S
States: *opened, toClose, toOpen, closed
Attributes:
Traits:
Attribute-Function: opened -> {};toClose -> {};toOpen ->
         {};closed -> {};
Parameter-Specifications:

Transition-Specifications:
     R1: <opened,toClose>; Lower[](true); true => true;
     R2: <toClose,closed>; Down[](true); true => true;
     R3: <toOpen,opened>; Up[](true); true => true;
     R4: <closed,toOpen>; Raise[](true); true => true;
Time-Constraints:
     TCvar1: R1, Down, [0, 1], {};
     TCvar2: R4, Up, [1, 2], {};
end
```

Figure 46: Gate TROM class – class specification

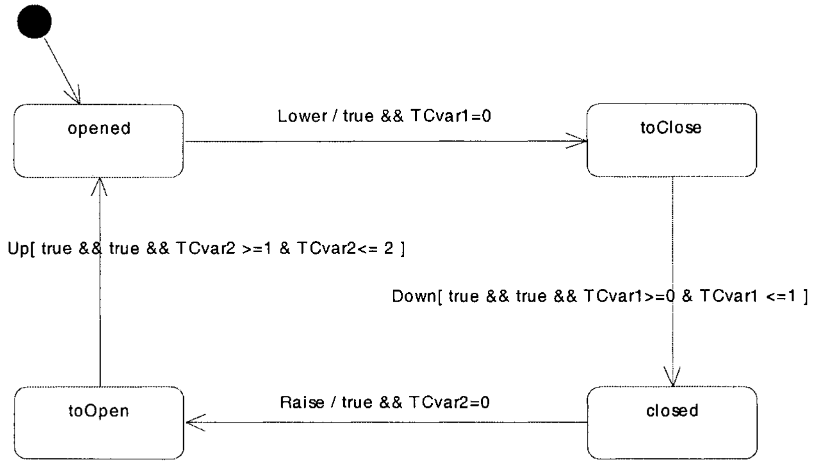

opened
Lower / true && TCvar1=0
toClose
Up[true && true && TCvar2 >=1 & TCvar2<= 2]
Down[true && true && TCvar1>=0 & TCvar1 <=1]
toOpen
Raise / true && TCvar2=0
closed

Figure 47: Gate TROM class – state chart diagram

6.3.5 Simulation Event List with Parameterized Events

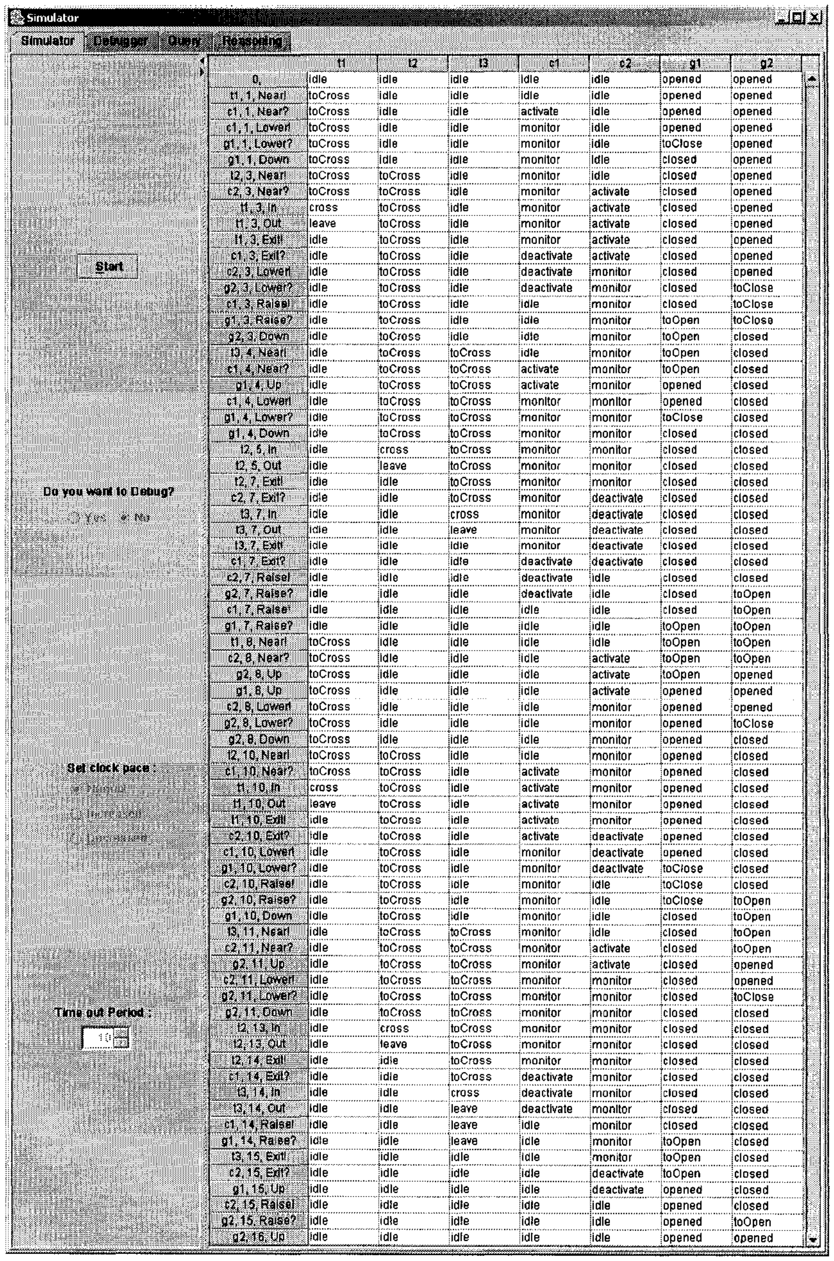
We use the same events list as in Section 3.3, except that the argument of the event *Near* carries the identification of the trains.

```
SEL: TCG
   t1, Near[TID=1], @C1, 1;
   t2, Near[TID=2], @C2, 3;
   t3, Near[TID=3], @C1, 4;
   t1, Near[TID=1], @C2, 8;
   t2, Near[TID=2], @C1, 10;
   t3, Near[TID=3], @C2, 11;
end
```

Figure 48: Sample Simulation Event List

6.3.6 Simulation Result with Parameterized Events

Figure 49 displays the simulation results with parameterized events. The simulator successfully modeled the entire interaction between several TROM objects, and it also verified that the revised Train-Gate-Controller model with parameterized events is correct.



Do you want to Debug? Yes [✓ No]

Set clock pace :

Time out Period : 10

	t1	t2	t3	c1	c2	g1	g2
0,	idle	idle	idle	idle	idle	opened	opened
t1, 1, Near!	toCross	idle	idle	idle	idle	opened	opened
c1, 1, Near?	toCross	idle	idle	activate	idle	opened	opened
c1, 1, Lower!	toCross	idle	idle	monitor	idle	opened	opened
g1, 1, Lower?	toCross	idle	idle	monitor	idle	toClose	opened
g1, 1, Down	toCross	idle	idle	monitor	idle	closed	opened
t2, 3, Near!	toCross	toCross	idle	monitor	idle	closed	opened
c2, 3, Near?	toCross	toCross	idle	monitor	activate	closed	opened
t1, 3, In	cross	toCross	idle	monitor	activate	closed	opened
t1, 3, Out	leave	toCross	idle	monitor	activate	closed	opened
t1, 3, Exit!	idle	toCross	idle	monitor	activate	closed	opened
c1, 3, Exit?	idle	toCross	idle	deactivate	activate	closed	opened
c2, 3, Lower!	idle	toCross	idle	deactivate	monitor	closed	opened
g2, 3, Lower?	idle	toCross	idle	deactivate	monitor	closed	toClose
c1, 3, Raise!	idle	toCross	idle	idle	monitor	closed	toClose
g1, 3, Raise?	idle	toCross	idle	idle	monitor	toOpen	toClose
g2, 3, Down	idle	toCross	idle	idle	monitor	toOpen	closed
t3, 4, Near!	idle	toCross	toCross	idle	monitor	toOpen	closed
c1, 4, Near?	idle	toCross	toCross	activate	monitor	toOpen	closed
g1, 4, Up	idle	toCross	toCross	activate	monitor	opened	closed
c1, 4, Lower!	idle	toCross	toCross	monitor	monitor	opened	closed
g1, 4, Lower?	idle	toCross	toCross	monitor	monitor	toClose	closed
g1, 4, Down	idle	toCross	toCross	monitor	monitor	closed	closed
t2, 5, In	idle	cross	toCross	monitor	monitor	closed	closed
t2, 5, Out	idle	leave	toCross	monitor	monitor	closed	closed
t2, 7, Exit!	idle	idle	toCross	monitor	monitor	closed	closed
c2, 7, Exit?	idle	idle	toCross	monitor	deactivate	closed	closed
t3, 7, In	idle	idle	cross	monitor	deactivate	closed	closed
t3, 7, Out	idle	idle	leave	monitor	deactivate	closed	closed
t3, 7, Exit!	idle	idle	idle	monitor	deactivate	closed	closed
c1, 7, Exit?	idle	idle	idle	deactivate	deactivate	closed	closed
c2, 7, Raise!	idle	idle	idle	deactivate	idle	closed	closed
g2, 7, Raise?	idle	idle	idle	deactivate	idle	closed	toOpen
c1, 7, Raise!	idle	idle	idle	idle	idle	closed	toOpen
g1, 7, Raise?	idle	idle	idle	idle	idle	toOpen	toOpen
t1, 8, Near!	toCross	idle	idle	idle	idle	toOpen	toOpen
c2, 8, Near?	toCross	idle	idle	idle	activate	toOpen	toOpen
g2, 8, Up	toCross	idle	idle	idle	activate	toOpen	opened
g1, 8, Up	toCross	idle	idle	idle	activate	opened	opened
c2, 8, Lower!	toCross	idle	idle	idle	monitor	opened	opened
g2, 8, Lower?	toCross	idle	idle	idle	monitor	opened	toClose
g2, 8, Down	toCross	idle	idle	idle	monitor	opened	closed
t2, 10, Near!	toCross	toCross	idle	idle	monitor	opened	closed
c1, 10, Near?	toCross	toCross	idle	activate	monitor	opened	closed
t1, 10, In	cross	toCross	idle	activate	monitor	opened	closed
t1, 10, Out	leave	toCross	idle	activate	monitor	opened	closed
t1, 10, Exit!	idle	toCross	idle	activate	monitor	opened	closed
c2, 10, Exit?	idle	toCross	idle	activate	deactivate	opened	closed
c1, 10, Lower!	idle	toCross	idle	monitor	deactivate	opened	closed
g1, 10, Lower?	idle	toCross	idle	monitor	deactivate	toClose	closed
c2, 10, Raise!	idle	toCross	idle	monitor	idle	toClose	closed
g2, 10, Raise?	idle	toCross	idle	monitor	idle	toClose	toOpen
g1, 10, Down	idle	toCross	idle	monitor	idle	closed	toOpen
t3, 11, Near!	idle	toCross	toCross	monitor	idle	closed	toOpen
c2, 11, Near?	idle	toCross	toCross	monitor	activate	closed	toOpen
g2, 11, Up	idle	toCross	toCross	monitor	activate	closed	opened
c2, 11, Lower!	idle	toCross	toCross	monitor	monitor	closed	opened
g2, 11, Lower?	idle	toCross	toCross	monitor	monitor	closed	toClose
g2, 11, Down	idle	toCross	toCross	monitor	monitor	closed	closed
t2, 13, In	idle	cross	toCross	monitor	monitor	closed	closed
t2, 13, Out	idle	leave	toCross	monitor	monitor	closed	closed
t2, 14, Exit!	idle	idle	toCross	monitor	monitor	closed	closed
c1, 14, Exit?	idle	idle	toCross	deactivate	monitor	closed	closed
t3, 14, In	idle	idle	cross	deactivate	monitor	closed	closed
t3, 14, Out	idle	idle	leave	deactivate	monitor	closed	closed
c1, 14, Raise!	idle	idle	leave	idle	monitor	closed	closed
g1, 14, Raise?	idle	idle	leave	idle	monitor	toOpen	closed
t3, 15, Exit!	idle	idle	idle	idle	monitor	toOpen	closed
c2, 15, Exit?	idle	idle	idle	idle	deactivate	toOpen	closed
g1, 15, Up	idle	idle	idle	idle	deactivate	opened	closed
c2, 15, Raise!	idle	idle	idle	idle	idle	opened	closed
g2, 15, Raise?	idle	idle	idle	idle	idle	opened	toOpen
g2, 16, Up	idle	idle	idle	idle	idle	opened	opened

Figure 49: The Simulation Result of TGC modeling with Parameterized Events

Chapter 7

Robotic Assembly Case Study

This chapter reviews the design of the robotics case. We will simulate the robotics system with our upgraded validation tool and expose the flaws in the design. Later, we will offer two solutions based on our analysis, which demonstrates the power of the validation tool. Finally, we will model the robotics system with parameterized events.

7.1 Problem Description

An assembly unit consists of a user, a conveyor belt, a vision system, a robot with two arms, and a tray for assembling. The user will place two kinds of parts, a dish and a cup, onto the conveyor belt. The belt will convey the parts towards the vision system. Whenever a part enters the sensor zone, the vision system will detect it and inform the belt to stop immediately. Next, the vision system will recognize the type of the part and communicate that to the robot, so that the robot can pick it up from the stopped conveyor belt. After the robot picks up a part, the belt resumes moving. An assembly will be finished when a dish and a cup are separately placed in the tray by two arms of the robot.

7.2 Assumptions and Time Constraints

The underlying assumptions are as follows:

- Both robotic arms share identical mechanical characteristics, including speed and angle.
- The algorithms for parts recognition, collision-free motion of the robotic arms, gripping, holding, and placement work in real-time.
- The conveyor belt, on which the parts are placed in sequence, moves at a constant speed. The parts on the conveyor belt will not collide with each other.

- The vision system has a sensor for detecting parts on the conveyor belt.

The entire assembly procedure must obey the following time constraints:

- Once a part enters the sensor zone, the sensor must detect the part within 2 time units.
- After the sensor has detected the part, the vision system must recognize the part and communicate the part type to the robot within 5 time units.
- Next, the robot must pick up the part from the conveyor belt within 2 time units of receiving the part type signal from the vision system.
- To complete an assembly, the right robotic arm should place the part it is holding onto the tray between 1 to 2 time units after it had picked up the part from the conveyor belt.
- Finally, after assembly has been completed, the assembled part must be removed from the tray between 1 to 2 time units.

7.3 Original Modeling

The assembly unit is abstracted as the following components: user, belt, vision system, robot, and tray. Figure 50 shows the TROM classes with relative port types and data types in the robotics system. Each component is modeled as a GRC with port types and attributes.

7.3.1 Class Diagram for Robotics System

1) Vision System TROM class is an aggregate of port types @U, @S, @Q
2) User TROM class is an aggregate of port types @VS
3) Belt TROM class is an aggregate of port types @R, @V
4) Robot TROM class is an aggregate of port types @C, @D, @E
5) Tray TROM class is an aggregate of port types @W

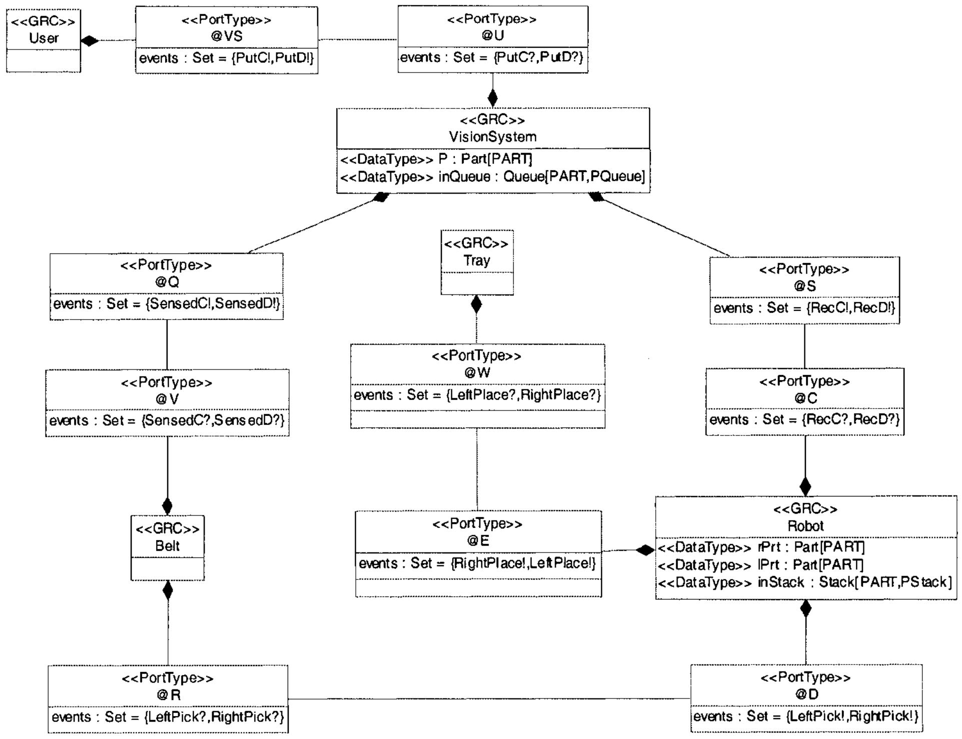

<<GRC>>
User
<<PortType>>
@VS
events : Set = {PutC!,PutD!}
<<PortType>>
@U
events : Set = {PutC?,PutD?}
<<GRC>>
VisionSystem
<<DataType>> P : Part[PART]
<<DataType>> inQueue : Queue[PART,PQueue]
<<PortType>>
@Q
events : Set = {SensedC!,SensedD!}
<<GRC>>
Tray
<<PortType>>
@S
events : Set = {RecC!,RecD!}
<<PortType>>
@W
events : Set = {LeftPlace?,RightPlace?}
<<PortType>>
@V
events : Set = {SensedC?,SensedD?}
<<PortType>>
@C
events : Set = {RecC?,RecD?}
<<GRC>>
Belt
<<PortType>>
@E
events : Set = {RightPlace!,LeftPlace!}
<<GRC>>
Robot
<<DataType>> rPrt : Part[PART]
<<DataType>> lPrt : Part[PART]
<<DataType>> inStack : Stack[PART,PStack]
<<PortType>>
@R
events : Set = {LeftPick?,RightPick?}
<<PortType>>
@D
events : Set = {LeftPick!,RightPick!}

Figure 50: Robotics System Class Diagram

The link between the port type @VS of the user and the port type @U of the vision system means that the user uses port @VS to communicate with Vision System through its port @U.

The link between the port type @Q of the vision system and the port type @V ofthe belt means that the vision system uses port @Q to communicate with the belt through its port @V.

The link between the port type @S of the vision system and the port type @C of the robot means that vision system uses port @S to communicate with tobot through its port @C.

The link between the port type @D of the robot and the port type @R of the belt means that the robot uses port @D to communicate with the belt through its port @R.

The link between the port type @E of the robot and the port type @W of the tray means that the robot uses port @E to communicate with the tray through its port @W.

The vision system has two attributes: P is a type of trait Part, and inQueue is a type of trait Queue. These two types are defined as LSL traits: Part and Queue have a parameter of Part type.

The robot has three attributes. Two of them are rPrt and lPrt that are a type of trait Part. The last one is inStack of trait Stack.

7.3.2 Formal Problem Description

Each class abstracted from the problem will be described with a state chart diagram, a class specification generated by the translator and used by the interpreter, and a class diagram. All abstract data types used in the robotics system will be defined as LSL traits, and all instantiations as well as communications among them are defined in the subsystem configuration specification (SCS).

User Class

The user is the only environmental class in the system, and all its output events cannot be constrained. This means that the user may place parts on the conveyor belt at any time, and the robotics system should have the capability to handle arbitrary events from the user.

```
Class User [@VS]
      Events: Next, PutD!@VS, PutC!@VS, Resume
      States: *idle, ready, place
      Attributes:
      Traits:
      Attribute-Function: idle -> {};ready -> {};place -> {};
      Transition-Specifications:
           R1: <idle,ready>; Next(true); true => true;
           R2: <ready,place>; PutD(true); true => true;
           R3: <ready,place>; PutC(true); true => true;
           R4: <place,idle>; Resume(true); true => true;
      Time-Constraints:

end
```

Figure 51: User TROM class – Class specification

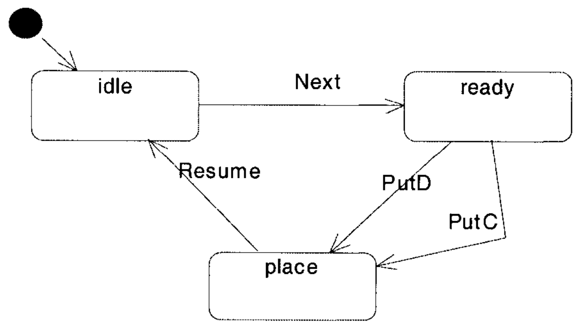


Figure 52: User TROM class – State chart diagram

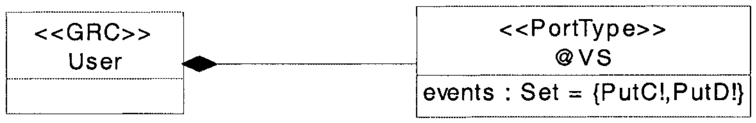


Figure 53: User TROM class – Class diagram

The Vision System Class

The user sends a message to the vision system when it places a part on the conveyor belt. Once the vision system receives the message from the user, the vision system records this event information in a Queue by inserting the part into the inQueue. Within a certain time frame, the vision system will will sense the part and inform the belt to stop, upon which the vision system will start recognizing the part. Once the vision system has recognized the part, it will signal the robot to pick it up. The belt will not resume moving until the robot has picks up the part and inform the belt. The vision system will record all messages from user into the queue during the entire assembly procedure.

```
Class VisionSystem [@U, @Q, @S]
Events: PutD?@U, PutC?@U, SensedD!@Q, SensedC!@Q, RecD!@S,
        RecC!@S
States: *Monitor, active, identify
Attributes: P:PART;inQueue:PQueue
Traits: Part[PART],Queue[PART,PQueue]
Attribute-Function: Monitor -> {inQueue};active ->
        {inQueue};identify -> {inQueue};
Transition-Specifications:
     R1: <Monitor,active>; PutD(true); true =>
        inQueue'=append(dish(),inQueue);
     R2: <Monitor,active>; PutC(true); true =>
        inQueue'=append(cup(),inQueue);
     R3: <active,active>; PutC(true); true =>
        inQueue'=append(cup(),inQueue);
     R4: <active,identify>; SensedD(true); head(inQueue)=dish()
        => true;
     R5: <active,identify>; SensedC(true); head(inQueue)=cup()
        => true;
     R6: <active,active>; PutD(true); true =>
        inQueue'=append(dish(),inQueue);
     R7: <identify,active>; RecD(true); len(inQueue)>1 =>
        inQueue'=tail(inQueue);
     R8: <identify,active>; RecC(true); len(inQueue)>1 =>
        inQueue'=tail(inQueue);
     R9: <identify,Monitor>; RecC(true); len(inQueue)=1 =>
        inQueue'=tail(inQueue);
     R10: <identify,Monitor>; RecD(true); len(inQueue)=1 =>
        inQueue'=tail(inQueue);
     R11: <identify,identify>; PutD(true); true =>
        inQueue'=append(dish(),inQueue);
     R12: <identify,identify>; PutC(true); true =>
        inQueue'=append(cup(),inQueue);
Time-Constraints:
     TCVar1: R1, SensedD, [0, 2], {};
     TCVar7: R7, SensedD, [0, 2], {};
     TCVar8: R8, SensedD, [0, 2], {};
     TCVar2: R2, SensedC, [0, 2], {};
     TCVar9: R7, SensedC, [0, 2], {};
     TCVar10: R8, SensedC, [0, 2], {};
     TCVar3: R4, RecD, [0, 5], {};
     TCVar4: R5, RecC, [0, 5], {};
     TCVar6: R5, RecC, [0, 5], {};
     TCVar5: R4, RecD, [0, 5], {};
end
```

Figure 54: Vision System TROM class – Class specification

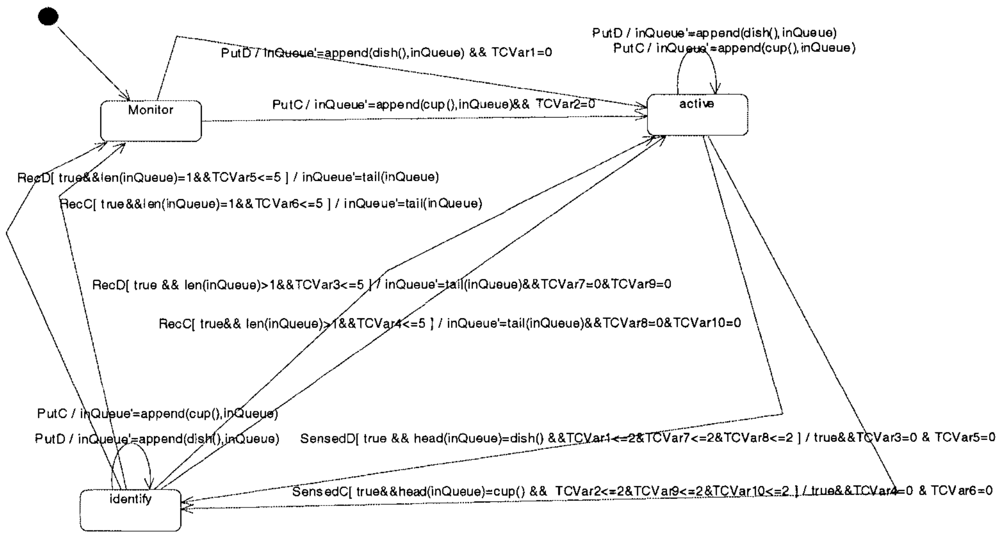

PutD / inQueue'=append(dish(),inQueue)
PutC / inQueue'=append(cup(),inQueue)
PutD / inQueue'=append(dish(),inQueue) && TCVar1=0
Monitor
PutC / inQueue'=append(cup(),inQueue)&& TCVar2=0
active
RecD[true&&len(inQueue)=1&&TCVar5<=5] / inQueue'=tail(inQueue)
RecC[true&&len(inQueue)=1&&TCVar6<=5] / inQueue'=tail(inQueue)
RecD[true && len(inQueue)>1&&TCVar3<=5] / inQueue'=tail(inQueue)&&TCVar7=0&TCVar9=0
RecC[true&& len(inQueue)>1&&TCVar4<=5] / inQueue'=tail(inQueue)&&TCVar8=0&TCVar10=0
PutC / inQueue'=append(cup(),inQueue)
PutD / inQueue'=append(dish(),inQueue)
SensedD[true && head(inQueue)=dish() &&TCVar1<=2&TCVar7<=2&TCVar8<=2] / true&&TCVar3=0 & TCVar5=0
SensedC[true&&head(inQueue)=cup() && TCVar2<=2&TCVar9<=2&TCVar10<=2] / true&&TCVar4=0 & TCVar6=0
identify

Figure 55: Vision System TROM class – State chart diagram

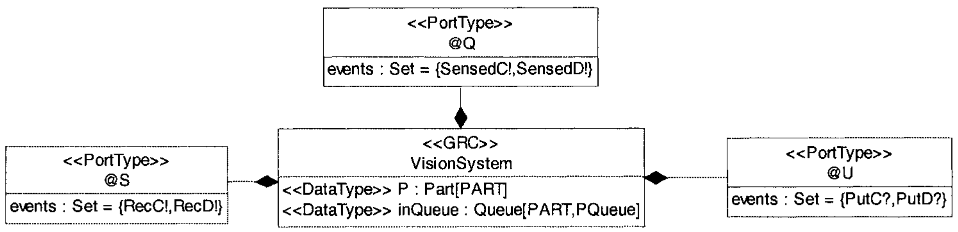

<<PortType>>
@Q
events : Set = {SensedC!,SensedD!}
<<PortType>>
@S
events : Set = {RecC!,RecD!}
<<GRC>>
VisionSystem
<<DataType>> P : Part[PART]
<<DataType>> inQueue : Queue[PART,PQueue]
<<PortType>>
@U
events : Set = {PutC?,PutD?}

Figure 56: Vision System TROM class – Class diagram

The Belt Class

The belt is controlled by the vision system and the robot. The belt stops whenever the vision system senses a part, and resumes moving whenever the robot picks up a part from the belt.

```
Class Belt [@V, @R]
Events: SensedC?@V, SensedD?@V, RightPick?@R, LeftPick?@R
States: *active, stop
Attributes:
Traits:
Attribute-Function: active -> {};stop -> {};
Transition-Specifications:
     R1: <active,stop>; SensedC(true); true => true;
     R2: <active,stop>; SensedD(true); true => true;
     R3: <stop,active>; RightPick(true); true => true;
     R4: <stop,active>; LeftPick(true); true => true;
Time-Constraints:
end
```

Figure 57: Belt TROM class – Class specification

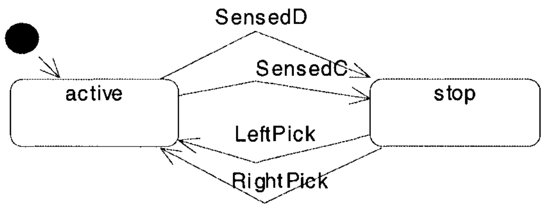


Figure 58: Belt TROM class – State chart diagram

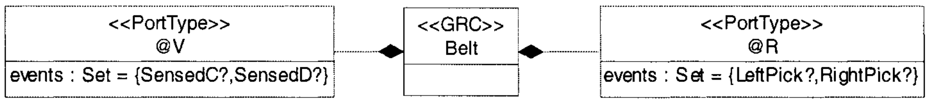

<<PortType>>
@V
events : Set = {SensedC?,SensedD?}
<<GRC>>
Belt
<<PortType>>
@R
events : Set = {LeftPick?,RightPick?}

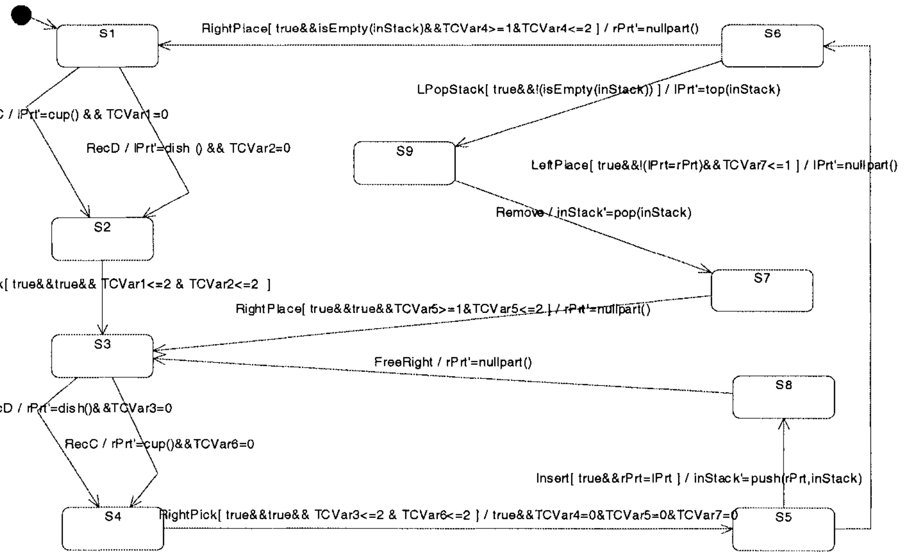
Figure 59: Belt TROM class – Class diagram

The Robot Class

The Robot has two arms: a left and a right arm. Whenever an arm picks up a part from the conveyor belt, the robot will signal the belt to resume moving. Whenever the robot receives a message from the vision system that a part is coming, the left arm, if it is empty, will pick up the part. If the left arm is full, the right arm will pick up the part. If the two robot arms are holding the same type of parts, the right arm will place the part it is holding onto the stack and wait for the next part on the belt. If both arms are holding different types of parts, the left arm will first place the part it is holding onto the tray. Next, the left arm will pick up a part from the stack, if the stack is not empty, or wait for the next part on the belt. Afterwards, the right arm will place the part that it is holding onto the tray to finish the assembly. If the stack contains no parts, the two arms will be free after the assembly; otherwise, only the right arm will be free.

```
Class Robot [@D, @C, @E]
Events: RecC?@C, RecD?@C, LeftPick!@D, RightPick!@D, Insert,
        LeftPlace!@E, RightPlace!@E, LPopStack, FreeRight,
        Remove
States: *S1, S2, S3, S4, S5, S6, S7, S8, S9
Attributes: rPrt:PART;lPrt:PART;inStack:PStack
Traits: Part[PART],Stack[PART,PStack]
Attribute-Function: S1 -> {rPrt};S2 -> {lPrt};S3 -> {rPrt};S4 -
         > {rPrt};S5 -> {};S6 -> {lPrt};S7 -> {inStack};S8 ->
         {inStack};S9 -> {lPrt};
Transition-Specifications:
     R1: <S1,S2>; RecC(true); true => lPrt'=cup();
     R2: <S1,S2>; RecD(true); true => lPrt'=dish();
     R3: <S2,S3>; LeftPick(true); true => true;
     R4: <S3,S4>; RecC(true); true => rPrt'=cup();
     R5: <S3,S4>; RecD(true); true => rPrt'=dish();
     R6: <S4,S5>; RightPick(true); true => true;
     R7: <S5,S8>; Insert(true); rPrt=lPrt =>
         inStack'=push(rPrt,inStack);
     R8: <S5,S6>; LeftPlace(true); !(lPrt=rPrt) =>
         lPrt'=nullpart();
     R9: <S6,S1>; RightPlace(true); isEmpty(inStack) =>
         rPrt'=nullpart();;
     R10: <S6,S9>; LPopStack(true); !(isEmpty(inStack)) =>
         lPrt'=top(inStack);
     R11: <S7,S3>; RightPlace(true); true => rPrt'=nullpart();
     R12: <S8,S3>; FreeRight(true); true => rPrt'=nullpart();
     R13: <S9,S7>; Remove(true); true => inStack'=pop(inStack);
Time-Constraints:
     TCVar1: R1, LeftPick, [0, 2], {};
     TCVar2: R2, LeftPick, [0, 2], {};
     TCVar3: R5, RightPick, [0, 2], {};
     TCVar6: R4, RightPick, [0, 2], {};
     TCVar7: R6, LeftPlace, [0, 1], {};
     TCVar4: R6, RightPlace, [1, 2], {};
     TCVar5: R6, RightPlace, [1, 2], {};
end
```

Figure 60: Robot TROM class – Class specification


S1
RightPlace[true&&isEmpty(inStack)&&TCVar4>=1&TCVar4<=2] / rPrt'=nullpart()
S6
LPopStack[true&&!(isEmpty(inStack))] / lPrt'=top(inStack)
RecC / lPrt'=cup() && TCVar1=0
RecD / lPrt'=dish () && TCVar2=0
S9
LeftPlace[true&&!(lPrt=rPrt)&&TCVar7<=1] / lPrt'=nullpart()
Remove / inStack'=pop(inStack)
S2
S7
LeftPick[true&&true&& TCVar1<=2 & TCVar2<=2]
RightPlace[true&&true&&TCVar5>=1&TCVar5<=2] / rPrt'=nullpart()
S3
FreeRight / rPrt'=nullpart()
S8
RecD / rPrt'=dish()&&TCVar3=0
RecC / rPrt'=cup()&&TCVar6=0
Insert[true&&rPrt=lPrt] / inStack'=push(rPrt,inStack)
S4
RightPick[true&&true&& TCVar3<=2 & TCVar6<=2] / true&&TCVar4=0&TCVar5=0&TCVar7=0
S5

S1 – Both arms are free
S2 – Left arm is ready to pick, right arm is free
S3 – Left arm is not free, right arm is free
S4 – Right arm is ready to pick, left arm is not free
S5 – Right arm is not free, left arm is not free
S6 – Left arm is free, right arm is not free
S7 – Right arm is ready to place, left arm is not free
S8 – Right arm is inserting into Stack, left arm is not free
S9 – Left arm removing from Stack, right arm is not free

Figure 61: Robot TROM class – State chart diagram

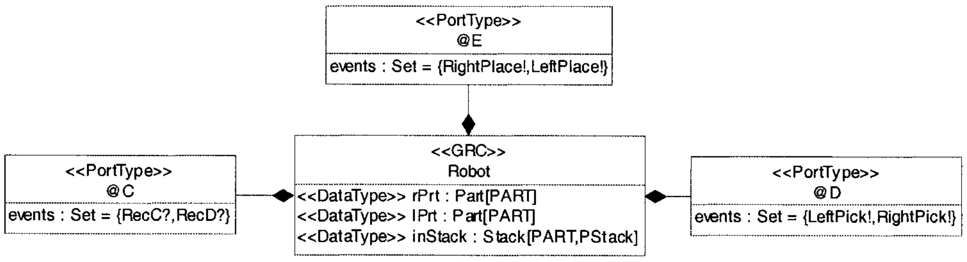

<<PortType>>
@E
events : Set = {RightPlace!,LeftPlace!}
<<GRC>>
Robot
<<DataType>> rPrt : Part[PART]
<<DataType>> lPrt : Part[PART]
<<DataType>> inStack : Stack[PART,PStack]
<<PortType>>
@C
events : Set = {RecC?,RecD?}
<<PortType>>
@D
events : Set = {LeftPick!,RightPick!}

Figure 62: Robot TROM class – Class diagram

The Tray Class

The tray assembles the parts after the robot places a set of different type of parts onto it.

```
Class Tray [@W]
Events: LeftPlace?@W, RightPlace?@W, Trash
States: *idle, wait, loading
Attributes:
 Traits:
 Attribute-Function: idle -> {};wait -> {};loading -> {};
 Transition-Specifications:
      R1: <idle,wait>; LeftPlace(true); true => true;
      R2: <wait,loading>; RightPlace(true); true => true;
      R3: <loading,idle>; Trash(true); true => true;
 Time-Constraints:
      TCVar1: R2, Trash, [1, 2], {};
end
```

Figure 63: Tray TROM class – Class specification

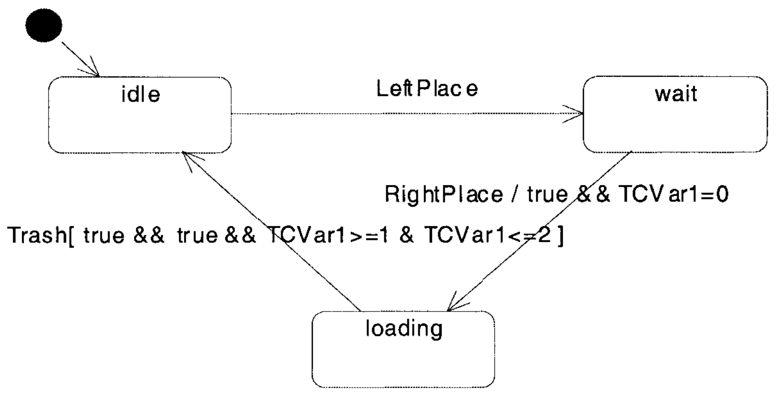

idle
LeftPlace
wait
RightPlace / true && TCVar1=0
Trash[true && true && TCVar1>=1 & TCVar1<=2]
loading

Figure 64: Tray TROM class – State chart diagram

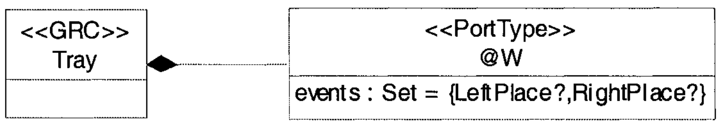


Figure 65: Tray TROM class – Class diagram

The Subsystem Configuration Specification (SCS)

The simulator will model a system that consists of one user, one belt, one vision system, one robot, and one tray. Figures 66 and 67 shows the system configuration specifications and collaboration diagram. These describe the specific instantiations and their communications in the instantiation of the Robotics system.

```
SCS BasicRobot
     Includes:
     Instantiate:
          U1::User[@VS:1];
          V1::VisionSystem[@U:1, @Q:1, @S:1];
          B1::Belt[@V:1, @R:1];
          R1::Robot[@D:1, @C:1, @E:1];
          T1::Tray[@W:1];
     Configure:
```

```
                V1.@U1:@U <-> U1.@VS1:@VS;
                B1.@V1:@V <-> V1.@Q1:@Q;
                R1.@C1:@C <-> V1.@S1:@S;
                R1.@E1:@E <-> T1.@W1:@W;
                B1.@R1:@R <-> R1.@D1:@D;
end
```

Figure 66: SCS – Specification

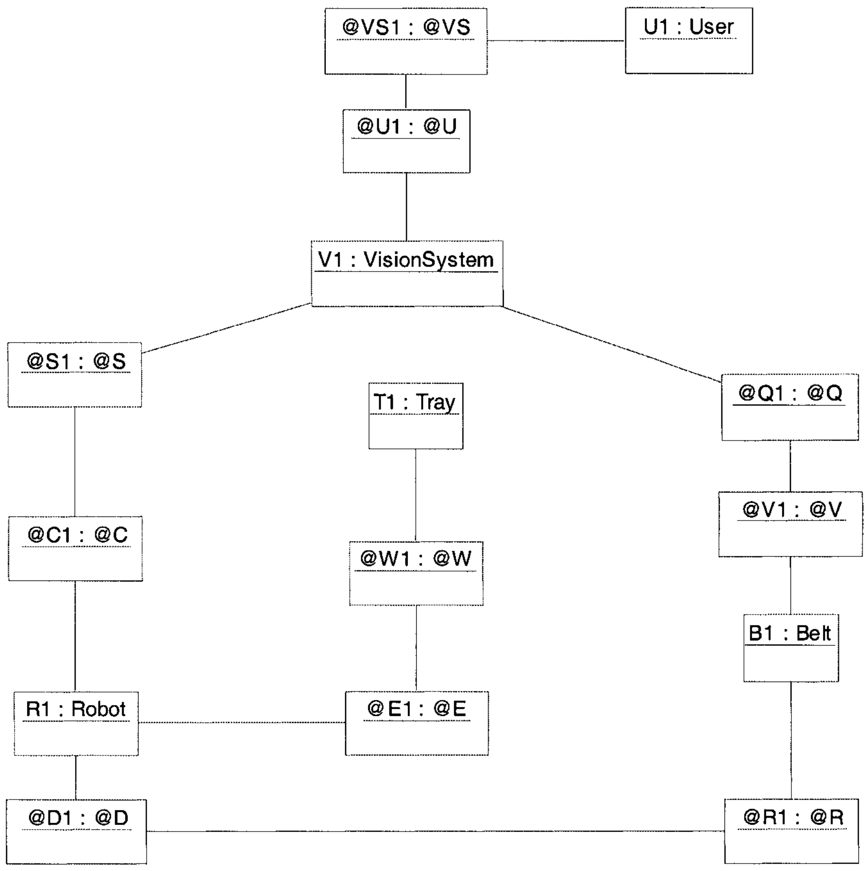


Figure 67: SCS – Collaboration diagram

LSL Traits

The system uses three LSL traits: part, queue, and stack. The vision system and the robot use the part trait. The vision system uses the queue trait to store events about coming parts on the belt. The robot uses the stack trait to store parts when the two hands are holding the same type of parts. Figures 68, 69, and 70 show the class specification of the three traits – part, queue, and stack.

```
Trait: Part(P)
     Includes:   Boolean
```

```
            Introduce:
                  cup :          -> P;
                  dish :         -> P;
                  free :         -> P;
      end
```

Figure 68: Part LSL Trait

```
Trait: Queue(e, Q)
Includes:   Integer, Boolean
Introduce:
      insert:     e, Q -> Q;
      delete:     Q -> Q;
      head:       Q -> e;
      size:       Q -> Int;
end
```

Figure 69: Queue LSL Trait

```
Trait: Stack(e, S)
Includes:   Boolean
Introduce:
      isEmpty:    S -> Bool;
      push:       e, S -> S;
      pop:        S -> S;
      top:        S -> e;
end
```

Figure 70: Stack LSL Trait

Sample Simulation Event List

This event list is used to simulate the external events of the robotics system. Four events, namely PutD, PutD, PutC and PutC of the User object, are instantiated in the SCS. All subsequent events are the responseto these stimuli.

```
SEL: BasicRobot
      PutD, U1, @VS1, 3;
      PutD, U1, @VS1, 5;
      PutC, U1, @VS1, 7;
      PutC, U1, @VS1, 9;
end
```

Figure 71: Sample Simulation Event List

7.3.3 Simulation Result Analysis

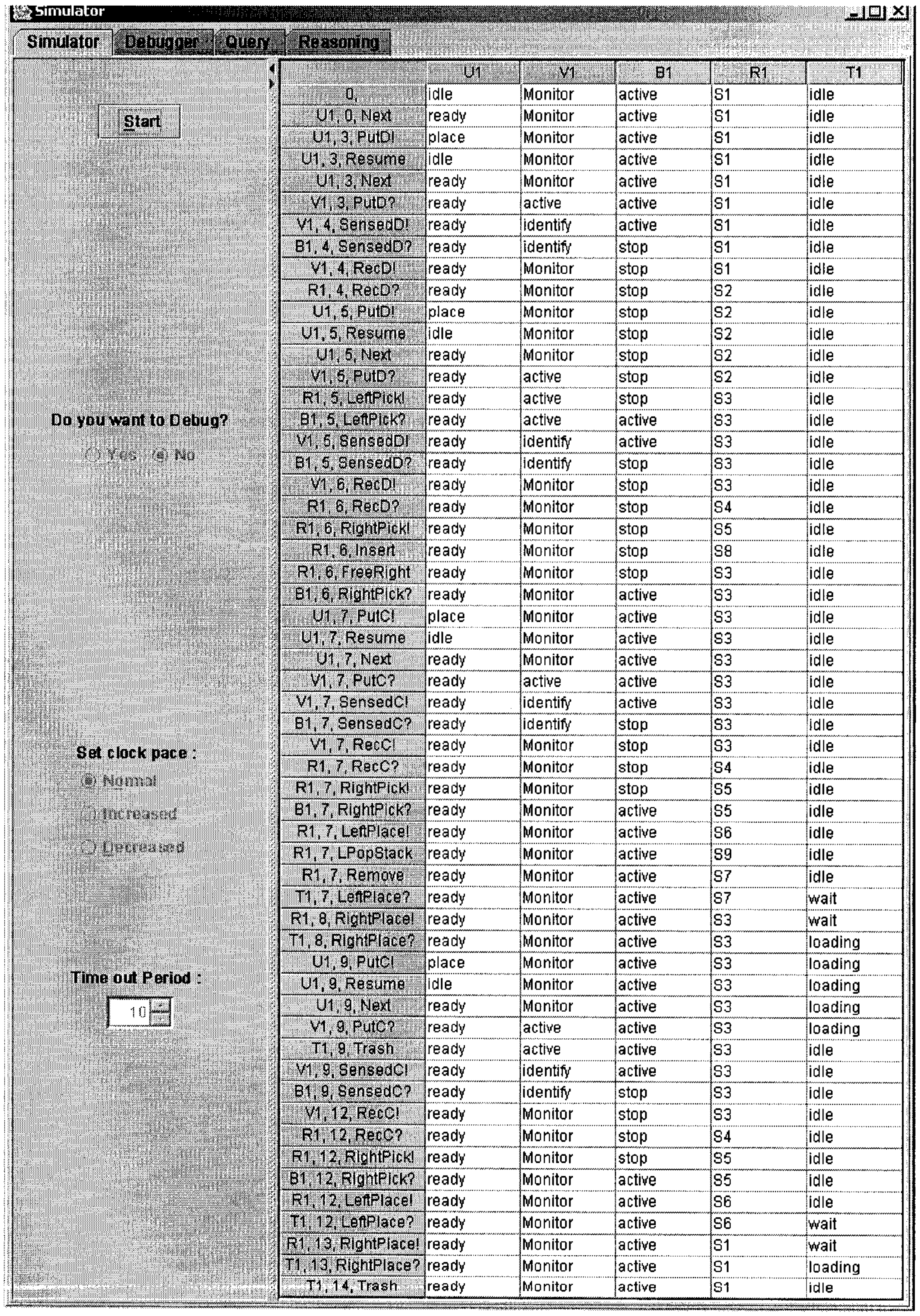
When we simulated the robotics in the simulator, we obtained amazing results. The simulator sometimes succeeded, such as in Figure 72; however, it sometimes failed, such as in Figure 73.

7.3.4 Explanation of the Result

In the table of Figure 72, the first column lists an object, the time at which it sent or received a message, and the event type, with "!" denoting an output event, "?" denoting an input event, and no symbol present denoting an internal event. The remaining columns, starting from the second column and going from left to right, display the state of the user object (U1), the vision system object (V1), the robot object (R1), the belt object (B1), and the tray object (T1), respectively, at the time specified in the first column. For example, the first row shows that when the system was initialized at 0 time unit, U1 was in an idle state, V1 was in a monitor state, B1 was in active state, R1 was in an S1 state, and T1 was in an idle state. The second row shows that when U1 issued a *Next* signal - an internal event – at 0 time units, U1 moved to a ready state. The third row shows that at the 3rd time unit, when U1 sent a PutD signal, meaning that it placed a dish on the belt, U1 moved to a place state. The sixth row shows that V1 moves to an active state after it receives the event – PutD from the User at the 3rd time unit. The seventh row shows that V1 sends the event SensedD and moves to the identity state at the 4th time unit.

7.3.5 Existing Problems in the Design

A careful analysis of Figure 73 uncovers some design problems. The next subsections will discuss them.



Do you want to Debug?

Yes [✓ No]

Set clock pace :

[✓ Normal] Increased Decreased

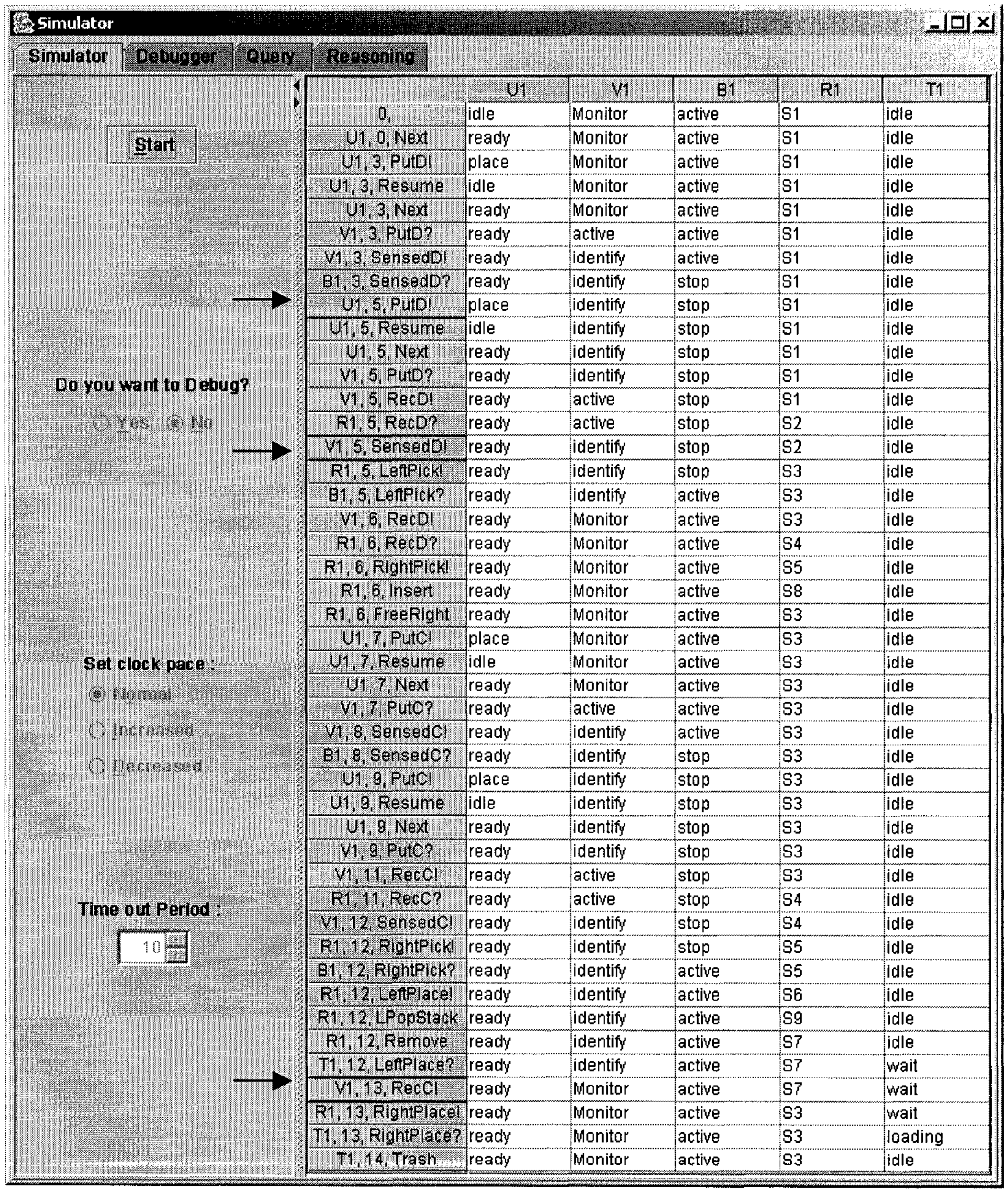
Time out Period : 10

	U1	V1	B1	R1	T1
0,	idle	Monitor	active	S1	idle
U1, 0, Next	ready	Monitor	active	S1	idle
U1, 3, PutD!	place	Monitor	active	S1	idle
U1, 3, Resume	idle	Monitor	active	S1	idle
U1, 3, Next	ready	Monitor	active	S1	idle
V1, 3, PutD?	ready	active	active	S1	idle
V1, 4, SensedD!	ready	identify	active	S1	idle
B1, 4, SensedD?	ready	identify	stop	S1	idle
V1, 4, RecD!	ready	Monitor	stop	S1	idle
R1, 4, RecD?	ready	Monitor	stop	S2	idle
U1, 5, PutD!	place	Monitor	stop	S2	idle
U1, 5, Resume	idle	Monitor	stop	S2	idle
U1, 5, Next	ready	Monitor	stop	S2	idle
V1, 5, PutD?	ready	active	stop	S2	idle
R1, 5, LeftPick!	ready	active	stop	S3	idle
B1, 5, LeftPick?	ready	active	active	S3	idle
V1, 5, SensedD!	ready	identify	active	S3	idle
B1, 5, SensedD?	ready	identify	stop	S3	idle
V1, 6, RecD!	ready	Monitor	stop	S3	idle
R1, 6, RecD?	ready	Monitor	stop	S4	idle
R1, 6, RightPick!	ready	Monitor	stop	S5	idle
R1, 6, Insert	ready	Monitor	stop	S8	idle
R1, 6, FreeRight	ready	Monitor	stop	S3	idle
B1, 6, RightPick?	ready	Monitor	active	S3	idle
U1, 7, PutC!	place	Monitor	active	S3	idle
U1, 7, Resume	idle	Monitor	active	S3	idle
U1, 7, Next	ready	Monitor	active	S3	idle
V1, 7, PutC?	ready	active	active	S3	idle
V1, 7, SensedC!	ready	identify	active	S3	idle
B1, 7, SensedC?	ready	identify	stop	S3	idle
V1, 7, RecC!	ready	Monitor	stop	S3	idle
R1, 7, RecC?	ready	Monitor	stop	S4	idle
R1, 7, RightPick!	ready	Monitor	stop	S5	idle
B1, 7, RightPick?	ready	Monitor	active	S5	idle
R1, 7, LeftPlace!	ready	Monitor	active	S6	idle
R1, 7, LPopStack	ready	Monitor	active	S9	idle
R1, 7, Remove	ready	Monitor	active	S7	idle
T1, 7, LeftPlace?	ready	Monitor	active	S7	wait
R1, 8, RightPlace!	ready	Monitor	active	S3	wait
T1, 8, RightPlace?	ready	Monitor	active	S3	loading
U1, 9, PutC!	place	Monitor	active	S3	loading
U1, 9, Resume	idle	Monitor	active	S3	loading
U1, 9, Next	ready	Monitor	active	S3	loading
V1, 9, PutC?	ready	active	active	S3	loading
T1, 9, Trash	ready	active	active	S3	idle
V1, 9, SensedC!	ready	identify	active	S3	idle
B1, 9, SensedC?	ready	identify	stop	S3	idle
V1, 12, RecC!	ready	Monitor	stop	S3	idle
R1, 12, RecC?	ready	Monitor	stop	S4	idle
R1, 12, RightPick!	ready	Monitor	stop	S5	idle
B1, 12, RightPick?	ready	Monitor	active	S5	idle
R1, 12, LeftPlace!	ready	Monitor	active	S6	idle
T1, 12, LeftPlace?	ready	Monitor	active	S6	wait
R1, 13, RightPlace!	ready	Monitor	active	S1	wait
T1, 13, RightPlace?	ready	Monitor	active	S1	loading
T1, 14, Trash	ready	Monitor	active	S1	idle

Figure 72: The successful simulation result of original modeling

User puts parts onto the Belt when the Belt stops

In Figure 73, refer to the first cell marked by a black arrow. Here, U1 sends a PutD event at the 5th time unit, meaning that U1 placed a dish on the belt. However, the belt is in a stop state at this time and cannot accept any more parts.



	U1	V1	B1	R1	T1
0,	idle	Monitor	active	S1	idle
U1, 0, Next	ready	Monitor	active	S1	idle
U1, 3, PutD!	place	Monitor	active	S1	idle
U1, 3, Resume	idle	Monitor	active	S1	idle
U1, 3, Next	ready	Monitor	active	S1	idle
V1, 3, PutD?	ready	active	active	S1	idle
V1, 3, SensedD!	ready	identify	active	S1	idle
B1, 3, SensedD?	ready	identify	stop	S1	idle
U1, 5, PutD!	place	identify	stop	S1	idle
U1, 5, Resume	idle	identify	stop	S1	idle
U1, 5, Next	ready	identify	stop	S1	idle
V1, 5, PutD?	ready	identify	stop	S1	idle
V1, 5, RecD!	ready	active	stop	S1	idle
R1, 5, RecD?	ready	active	stop	S2	idle
V1, 5, SensedD!	ready	identify	stop	S2	idle
R1, 5, LeftPick!	ready	identify	stop	S3	idle
B1, 5, LeftPick?	ready	identify	active	S3	idle
V1, 6, RecD!	ready	Monitor	active	S3	idle
R1, 6, RecD?	ready	Monitor	active	S4	idle
R1, 6, RightPick!	ready	Monitor	active	S5	idle
R1, 6, Insert	ready	Monitor	active	S8	idle
R1, 6, FreeRight	ready	Monitor	active	S3	idle
U1, 7, PutC!	place	Monitor	active	S3	idle
U1, 7, Resume	idle	Monitor	active	S3	idle
U1, 7, Next	ready	Monitor	active	S3	idle
V1, 7, PutC?	ready	active	active	S3	idle
V1, 8, SensedC!	ready	identify	active	S3	idle
B1, 8, SensedC?	ready	identify	stop	S3	idle
U1, 9, PutC!	place	identify	stop	S3	idle
U1, 9, Resume	idle	identify	stop	S3	idle
U1, 9, Next	ready	identify	stop	S3	idle
V1, 9, PutC?	ready	identify	stop	S3	idle
V1, 11, RecC!	ready	active	stop	S3	idle
R1, 11, RecC?	ready	active	stop	S4	idle
V1, 12, SensedC!	ready	identify	stop	S4	idle
R1, 12, RightPick!	ready	identify	stop	S5	idle
B1, 12, RightPick?	ready	identify	active	S5	idle
R1, 12, LeftPlace!	ready	identify	active	S6	idle
R1, 12, LPopStack	ready	identify	active	S9	idle
R1, 12, Remove	ready	identify	active	S7	idle
T1, 12, LeftPlace?	ready	identify	active	S7	wait
V1, 13, RecC!	ready	Monitor	active	S7	wait
R1, 13, RightPlace!	ready	Monitor	active	S3	wait
T1, 13, RightPlace?	ready	Monitor	active	S3	loading
T1, 14, Trash	ready	Monitor	active	S3	idle

Figure 73: The failure simulation result of original modeling

The cell marked by the second black arrow shows that V1 issued a SensedD message at the 5th time unit, meaning that the vision system sensed the dish that U1 placed on the belt at the 5th time unit. The belt (B1) should then receive this event and stop; however, because the belt is already in a stop state, there is no transition, which is defined in the state machine of the belt, corresponding to the SensedD event in the stop state.

The Robot loses some events coming from the Vision System

The cell marked by the third black arrow shows that V1 issued a RecC message at the 13th time unit, meaning that the visualizing system finished the recognizing stage and told the robot to pick up the part. But, the expected event – R1, 13, RecC? – did not occur because the robot was in an S7 state, which means that the Robot just picked up the part and was going to place the part from its left hand to the tray. The event – RecC or RecD – did not trigger a transition in this state. In fact, when the robot placed parts onto the tray, it may have spent some time in states S4, S5, S6, and S7, whose outgoing transitions are triggered by output events. In the meantime, it is possible that the vision system senses and recognizes a part and sends an event to it.

7.3.6 Summary of problems

(1) The belt cannot accept an arbitrary number of parts at any time. When the belt is stopped, it cannot accept more parts; however, the user must be able to place parts on the belt at any time.

(2) The robot cannot accept messages when it is placing parts onto the tray. Because the belt will move again after the robot picks up a part, it is possible that the robot will receive a message about the next part while the robot is placing parts onto the tray.

7.4 Remodeling

The user's requirement that it be able to place parts on the belt at any time conflicted with the belt's limitation that it can only accept parts while it is moving. This tension caused the original design problems. To remedy this, two aspects of the system may be remodeled – the environment and the system itself. If the user is predictable, i.e., the User will place parts at a well-controlled speed, the system just needs a subtle change. However, if the user is unpredictable, the system must be good enough to adapt to this rigorous environment. Here are the two solutions.

7.4.1 Robotics System with a Self-controlled User

Assume that the system exists in an ideal environment. The only environmental object – the user – is a self-controlled worker, who will place a part on the belt only while the belt is active. Before remodeling, we need to calculate two time values. One is the amount of time units that the Belt is in stop state, i.e. from the instant the vision system senses a part and directs the belt to stop to the instant that the robot picks up the part and directs the belt to resume. The vision system will recognize this part within 5 time units after sensing it. The Robot will pick it up within 2 time units. So the maximum time units are 7 time units. Another calculation is the amount of time units between two RecC or RecD. In this period, the robot must finish all actions after receiving a RecC/RecD event and go back to S1 or S3 state (Figure 61). The robot will pick up a cup or dish within 2 time units after it receives a RecC or RecD event. Subsequently, if the right arm is holding the same type of part held in the left arm, the right arm will place this part onto the stack within 1 time unit. Alternatively, if the right arm picked up a different type of part than that held in the left arm, the robot will place both of them onto the tray within 2 time units. Therefore, the maximum is 4 time units, which consists of the right arm picking up a part (i.e., 2 time units) and both arms placing the parts that they are holding in the tray (i.e., another 2 time units). For solving the two problems mentioned in section 1.4.3, we make the following changes:

(1) From the moment the user places a part on the belt to moment the robot picks it up and instructs the belt to stop, a maximum of 7 time units may have elapsed.

- Change the time interval of SEL to 7 or more time units

(2) The vision system takes 4 to 5 time units to recognize a part to avoid sending a message during the robot placing parts and the tray handling them.
- Change the time constraint Rec to the a window of 4 to 5 time units

As a result, the User, the Belt, and the Vision System classes need to be changed.

The Modified Belt Class

We added an indicator, like a traffic light, to signal the user when it could place additional parts on the belt.

```
Class Belt [@V, @R, @H]
Events: SensedC?@V, SensedD?@V, LeftPick?@R, RightPick?@R,
         Start!@H
States: *active, stop, toActive
Attributes:
Traits:
Attribute-Function: active -> {};stop -> {};toActive -> {};
Transition-Specifications:
     R1: <active,stop>; SensedC(true); true => true;
     R2: <active,stop>; SensedD(true); true => true;
     R3: <stop,toActive>; LeftPick(true); true => true;
     R4: <stop,toActive>; RightPick(true); true => true;
     R5: <toActive,active>; Start(true); true => true;
Time-Constraints:
     TCVar1: R3, Start, [0, 1], {};
     TCVar2: R4, Start, [0, 1], {};
end
```

Figure 74: Modified Belt TROM class – Class specification

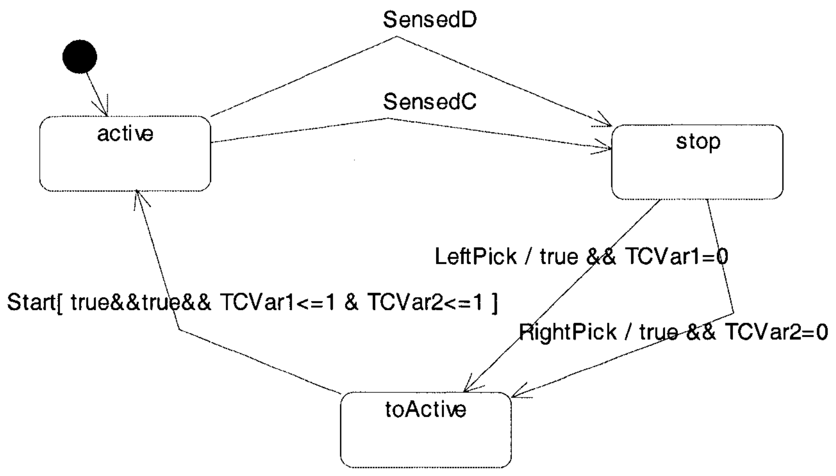

SensedD
SensedC
active
stop
LeftPick / true && TCVar1=0
Start[true&&true&& TCVar1<=1 & TCVar2<=1]
RightPick / true && TCVar2=0
toActive

Figure 75: Modified Belt TROM class – state chart diagram

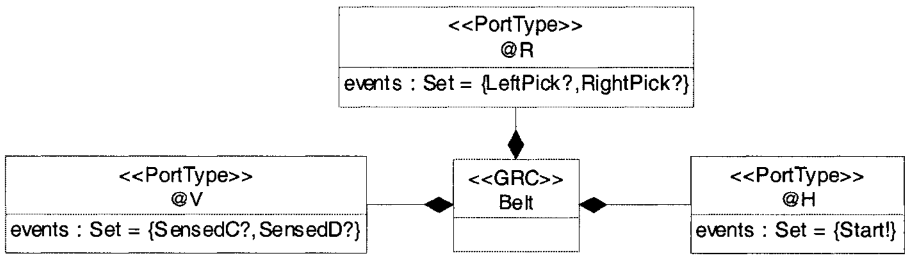


Figure 76: Modified Belt TROM class – class diagram

The Modified User Class

After the user places the first part on the belt, the user cannot place additional parts on the belt until it receives a signal from the belt that such is permissible.

```
Class User [@VS, @F]
Events: Next, PutD!@VS, PutC!@VS, Resume, Start?@F
States: *idle, ready, place, wait
Attributes:
Traits:
Attribute-Function: idle -> {};ready -> {};place -> {};wait ->
{};
Transition-Specifications:
     R1: <idle,ready>; Next(true); true => true;
     R2: <ready,place>; PutD(true); true => true;
     R3: <ready,place>; PutC(true); true => true;
     R4: <place,wait>; Start(true); true => true;
     R5: <wait,idle>; Resume(true); true => true;
Time-Constraints:

end
```

Figure 77: Modified User TROM class – Class specification

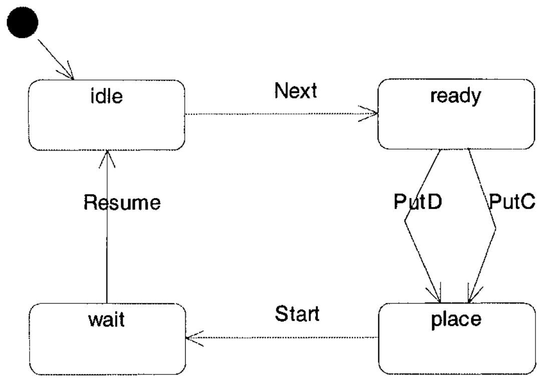


Figure 78: Modified User TROM class – State chart diagram

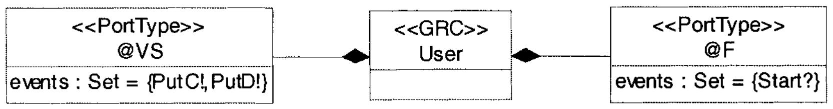


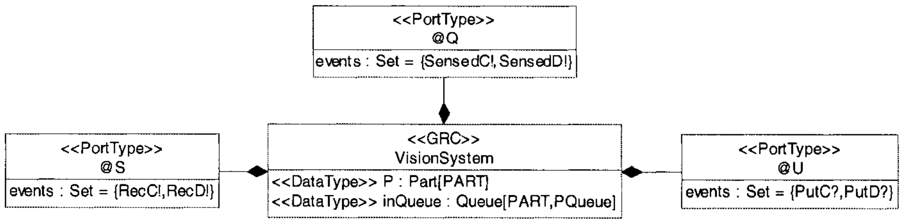
Figure 79: Modified User TROM class – Class diagram

The Modified Vision System Class

The time constraint of events RecC and RecD were changed for the vision system to the range [4, 5]. This avoids sending a recognition message to the robot during the robot placing parts onto the tray and the tray handling them.

```
Class VisionSystem [@U, @Q, @S]
Events: PutD?@U, PutC?@U, SensedD!@Q, SensedC!@Q, RecD!@S,
        RecC!@S
States: *Monitor, active, identify
Attributes: P:PART;inQueue:PQueue
Traits: Part[PART],Queue[PART,PQueue]
        Attribute-Function: Monitor -> {inQueue};active ->
        {inQueue};identify -> {inQueue};
Transition-Specifications:
     R1: <Monitor,active>; PutD(true); true =>
        inQueue'=append(dish(),inQueue);
     R2: <Monitor,active>; PutC(true); true =>
        inQueue'=append(cup(),inQueue);
     R3: <active,active>; PutC(true); true =>
        inQueue'=append(cup(),inQueue);
     R4: <active,identify>; SensedD(true); head(inQueue)=dish()
        => true;
     R5: <active,identify>; SensedC(true); head(inQueue)=cup()
        => true;
     R6: <active,active>; PutD(true); true =>
        inQueue'=append(dish(),inQueue);
     R7: <identify,active>; RecD(true); len(inQueue)>1 =>
        inQueue'=tail(inQueue);
     R8: <identify,active>; RecC(true); len(inQueue)>1 =>
        inQueue'=tail(inQueue);
     R9: <identify,Monitor>; RecC(true); len(inQueue)=1 =>
        inQueue'=tail(inQueue);
     R10: <identify,Monitor>; RecD(true); len(inQueue)=1 =>
        inQueue'=tail(inQueue);
     R11: <identify,identify>; PutD(true); true =>
        inQueue'=append(dish(),inQueue);
     R12: <identify,identify>; PutC(true); true =>
        inQueue'=append(cup(),inQueue);
Time-Constraints:
     TCVar1: R1, SensedD, [0, 2], {};
     TCVar7: R7, SensedD, [0, 2], {};
     TCVar8: R8, SensedD, [0, 2], {};
     TCVar2: R2, SensedC, [0, 2], {};
     TCVar9: R7, SensedC, [0, 2], {};
     TCVar10: R8, SensedC, [0, 2], {};
     TCVar3: R4, RecD, [4, 5], {};
     TCVar4: R5, RecC, [4, 5], {};
     TCVar6: R5, RecC, [4, 5], {};
     TCVar5: R4, RecD, [4, 5], {};
end
```

Figure 80: Modified Vision System class – Class specification



<<PortType>>
@Q
events : Set = {SensedC!, SensedD!}
<<PortType>>
@S
events : Set = {RecC!,RecD!}
<<GRC>>
VisionSystem
<<DataType>> P : Part[PART]
<<DataType>> inQueue : Queue[PART,PQueue]
<<PortType>>
@U
events : Set = {PutC?,PutD?}

Figure 81: Modified Vision System class – Class diagram

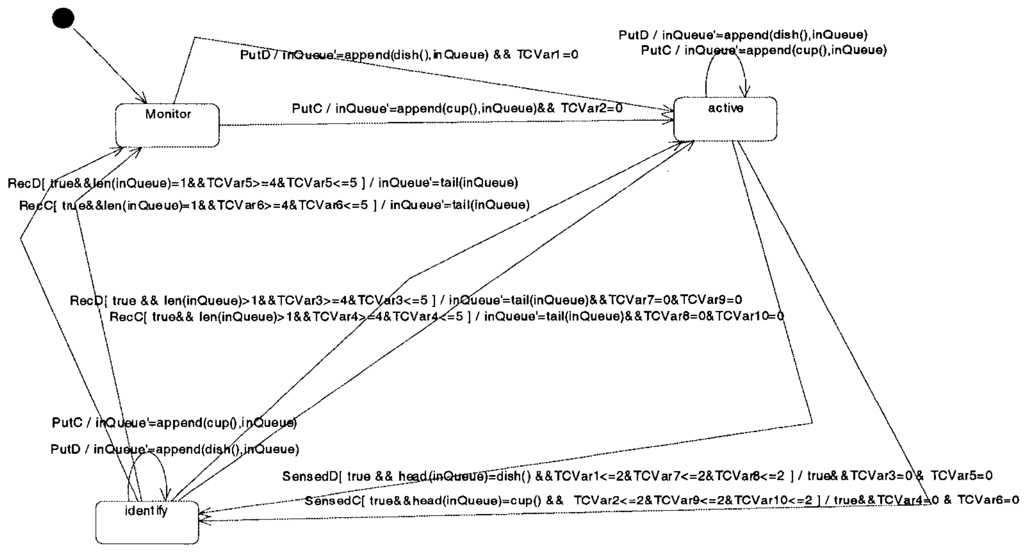

PutD / inQueue'=append(dish(),inQueue)
PutC / inQueue'=append(cup(),inQueue)
PutD / inQueue'=append(dish(),inQueue) && TCVar1=0
Monitor
PutC / inQueue'=append(cup(),inQueue)&& TCVar2=0
active
RecD[true&&len(inQueue)=1&&TCVar5>=4&TCVar5<=5] / inQueue'=tail(inQueue)
RecC[true&&len(inQueue)=1&&TCVar6>=4&TCVar6<=5] / inQueue'=tail(inQueue)
RecD[true && len(inQueue)>1&&TCVar3>=4&TCVar3<=5] / inQueue'=tail(inQueue)&&TCVar7=0&TCVar9=0
RecC[true&& len(inQueue)>1&&TCVar4>=4&TCVar4<=5] / inQueue'=tail(inQueue)&&TCVar8=0&TCVar10=0
PutC / inQueue'=append(cup(),inQueue)
PutD / inQueue'=append(dish(),inQueue)
SensedD[true && head(inQueue)=dish() &&TCVar1<=2&TCVar7<=2&TCVar8<=2] / true&&TCVar3=0& TCVar5=0
SensedC[true&&head(inQueue)=cup() && TCVar2<=2&TCVar9<=2&TCVar10<=2] / true&&TCVar4=0 & TCVar6=0
identify

Figure 82: Modified Vision System class – State chart diagram

The Sample Simulation Event List

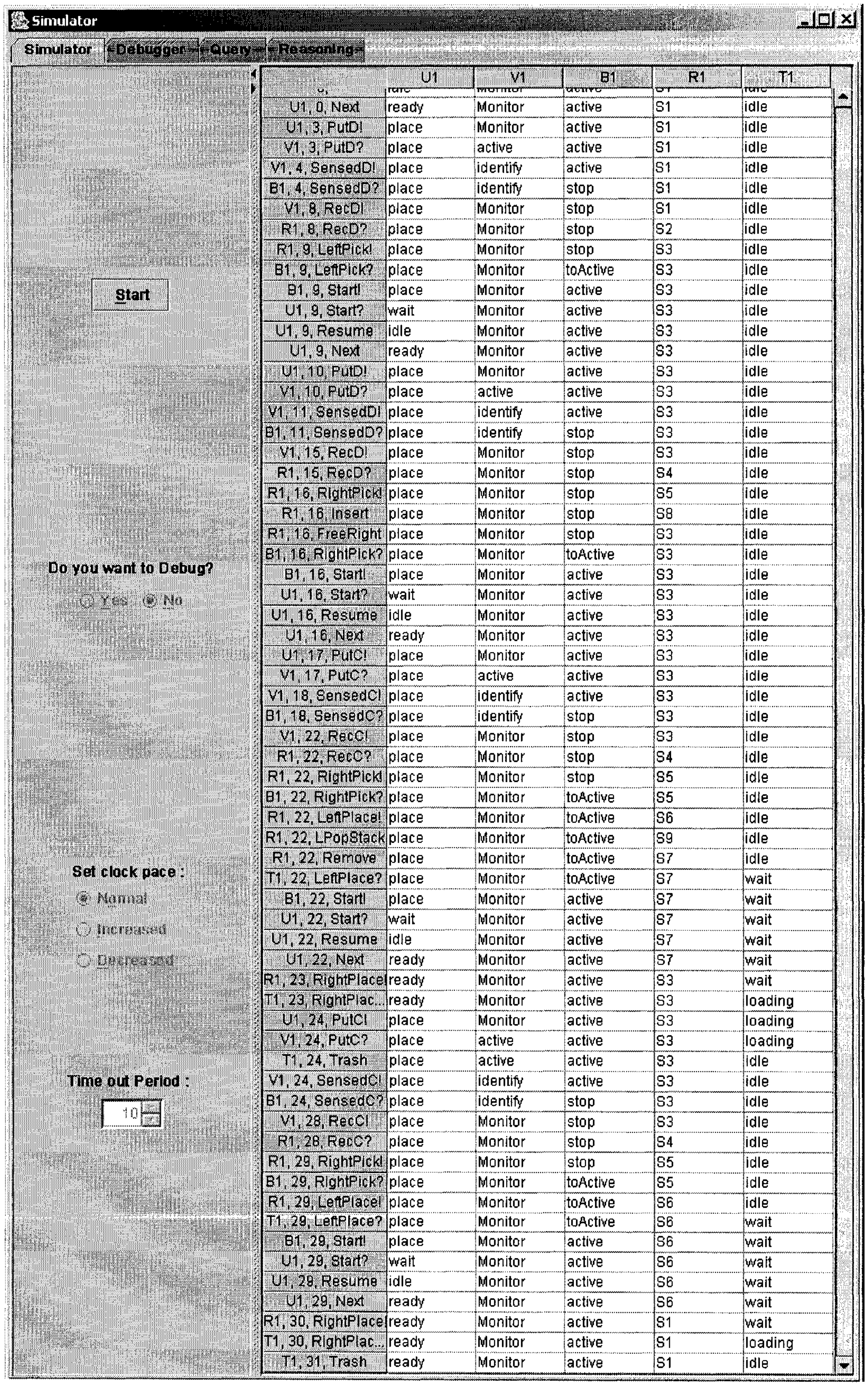
The interval of events was extended to 12 time units, so that one part will not be placed onto the belt until the previous one has been handled.

```
SEL: BasicRobot
          PutD, U1, @VS1, 3;
          PutD, U1, @VS1, 10;
          PutC, U1, @VS1, 17;
          PutC, U1, @VS1, 24;
end
```

Figure 83: Modified Sample Simulation Event List

The Simulation Result for the Modified Design

The simulator validated that this design is correct. Now, the User won't place parts when the belt stops, and the vision system won't send recognition messages when the robot is in S4, S5, S6, and S7 states. Figure 84 displays the simulation results.



Do you want to Debug?
Yes [✓ No]

Set clock pace :
[✓ Normal]
Increased
Decreased

Time out Period : 10

	U1	V1	B1	R1	T1
U1, 0, Next	ready	Monitor	active	S1	idle
U1, 3, PutD!	place	Monitor	active	S1	idle
V1, 3, PutD?	place	active	active	S1	idle
V1, 4, SensedD!	place	identify	active	S1	idle
B1, 4, SensedD?	place	identify	stop	S1	idle
V1, 8, RecD!	place	Monitor	stop	S1	idle
R1, 8, RecD?	place	Monitor	stop	S2	idle
R1, 9, LeftPick!	place	Monitor	stop	S3	idle
B1, 9, LeftPick?	place	Monitor	toActive	S3	idle
B1, 9, Start!	place	Monitor	active	S3	idle
U1, 9, Start?	wait	Monitor	active	S3	idle
U1, 9, Resume	idle	Monitor	active	S3	idle
U1, 9, Next	ready	Monitor	active	S3	idle
U1, 10, PutD!	place	Monitor	active	S3	idle
V1, 10, PutD?	place	active	active	S3	idle
V1, 11, SensedD!	place	identify	active	S3	idle
B1, 11, SensedD?	place	identify	stop	S3	idle
V1, 15, RecD!	place	Monitor	stop	S3	idle
R1, 15, RecD?	place	Monitor	stop	S4	idle
R1, 16, RightPick!	place	Monitor	stop	S5	idle
R1, 16, Insert	place	Monitor	stop	S8	idle
R1, 16, FreeRight	place	Monitor	stop	S3	idle
B1, 16, RightPick?	place	Monitor	toActive	S3	idle
B1, 16, Start!	place	Monitor	active	S3	idle
U1, 16, Start?	wait	Monitor	active	S3	idle
U1, 16, Resume	idle	Monitor	active	S3	idle
U1, 16, Next	ready	Monitor	active	S3	idle
U1, 17, PutC!	place	Monitor	active	S3	idle
V1, 17, PutC?	place	active	active	S3	idle
V1, 18, SensedC!	place	identify	active	S3	idle
B1, 18, SensedC?	place	identify	stop	S3	idle
V1, 22, RecC!	place	Monitor	stop	S3	idle
R1, 22, RecC?	place	Monitor	stop	S4	idle
R1, 22, RightPick!	place	Monitor	stop	S5	idle
B1, 22, RightPick?	place	Monitor	toActive	S5	idle
R1, 22, LeftPlace!	place	Monitor	toActive	S6	idle
R1, 22, LPopStack	place	Monitor	toActive	S9	idle
R1, 22, Remove	place	Monitor	toActive	S7	idle
T1, 22, LeftPlace?	place	Monitor	toActive	S7	wait
B1, 22, Start!	place	Monitor	active	S7	wait
U1, 22, Start?	wait	Monitor	active	S7	wait
U1, 22, Resume	idle	Monitor	active	S7	wait
U1, 22, Next	ready	Monitor	active	S7	wait
R1, 23, RightPlace!	ready	Monitor	active	S3	wait
T1, 23, RightPlac...	ready	Monitor	active	S3	loading
U1, 24, PutC!	place	Monitor	active	S3	loading
V1, 24, PutC?	place	active	active	S3	loading
T1, 24, Trash	place	active	active	S3	idle
V1, 24, SensedC!	place	identify	active	S3	idle
B1, 24, SensedC?	place	identify	stop	S3	idle
V1, 28, RecC!	place	Monitor	stop	S3	idle
R1, 28, RecC?	place	Monitor	stop	S4	idle
R1, 29, RightPick!	place	Monitor	stop	S5	idle
B1, 29, RightPick?	place	Monitor	toActive	S5	idle
R1, 29, LeftPlace!	place	Monitor	toActive	S6	idle
T1, 29, LeftPlace?	place	Monitor	toActive	S6	wait
B1, 29, Start!	place	Monitor	active	S6	wait
U1, 29, Start?	wait	Monitor	active	S6	wait
U1, 29, Resume	idle	Monitor	active	S6	wait
U1, 29, Next	ready	Monitor	active	S6	wait
R1, 30, RightPlace!	ready	Monitor	active	S1	wait
T1, 30, RightPlac...	ready	Monitor	active	S1	loading
T1, 31, Trash	ready	Monitor	active	S1	idle

Figure 84: Simulation Result of Self-controlled UserModeling

7.4.2 Robotics System with Buffered Belt

The user and belt, vision system and robot, and robot and tray have a sender and receiver-type relationship in a network transmission environment. If a sender transmits a signal at a lower speed than that which a receiver can handle, the receiver does not have to set-up a buffer; otherwise, the receiver must have a buffer to store the data which it could not handle at the time of the transmission. Similarly, if a user places parts on the belt at a rate faster than the belt can handle, the belt should have a buffer-like accessory to store these overflow parts. Likewise, the robot requires a buffer to store messages coming from the vision system while the robot is placing parts and cannot handle any messages. In addition, because the tray cannot handle more than one assembly at the same time, the robot only needs to adjust its time constraint to avoid placing too many parts in a short time. The new design alters the interaction between user, belt, and vision system. The user will send messages to the belt instead of the vision system, and the belt will send sensing messages to the vision system through its sensor. So, the vision system will not have to queue messages coming from the user because the vision system will handle only one part at any time. The new design includes the following modifications:

(1) Change port links among user, belt, and vision system.
(2) Add a queue to the belt to store overflow parts that placed by the user.
(3) Add a queue to the robot to store RecC/RecD events coming from the vision system
(4) The left arm of the Robot should place on the assembly pad the part it is holding within a 2 to 3 time units of picking up the part from the belt.
(5) The right arm of the robot should place on the assembly pad the part it is holding within 3 to 4 time units of picking up the part from the belt.
(6) Remove the queue from the vision system

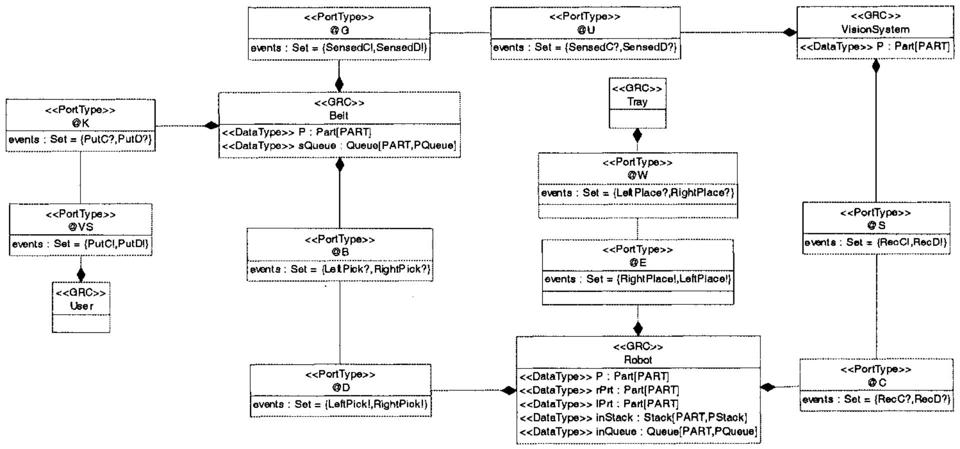
As a result, the system collaboration, the belt, the vision system, and the robot classes need to be changed.

New Class Diagram for Robotics Diagram

Because the interactions have been changed, the port definitions are modified as follows:

1) Vision System TROM class is an aggregate of port types @U, @S

2) User TROM class is an aggregate of port types @VS
3) Belt TROM class is an aggregate of port types @G, @K, @B
4) Robot TROM class is an aggregate of port types @C, @D, @E
5) Tray TROM class is an aggregate of port types @W



<<PortType>>
@G
events : Set = {SensedC!,SensedD!}
<<PortType>>
@U
events : Set = {SensedC?,SensedD?}
<<GRC>>
VisionSystem
<<DataType>> P : Part[PART]
<<PortType>>
@K
events : Set = {PutC?,PutD?}
<<GRC>>
Belt
<<DataType>> P : Part[PART]
<<DataType>> sQueue : Queue[PART,PQueue]
<<GRC>>
Tray
<<PortType>>
@W
events : Set = {LeftPlace?,RightPlace?}
<<PortType>>
@VS
events : Set = {PutC!,PutD!}
<<PortType>>
@B
events : Set = {LeftPick?,RightPick?}
<<PortType>>
@S
events : Set = {RecC!,RecD!}
<<PortType>>
@E
events : Set = {RightPlace!,LeftPlace!}
<<GRC>>
User
<<GRC>>
Robot
<<DataType>> P : Part[PART]
<<DataType>> rPrt : Part[PART]
<<DataType>> lPrt : Part[PART]
<<DataType>> inStack : Stack[PART,PStack]
<<DataType>> inQueue : Queue[PART,PQueue]
<<PortType>>
@D
events : Set = {LeftPick!,RightPick!}
<<PortType>>
@C
events : Set = {RecC?,RecD?}

Figure 85: Class diagram for Robotics system

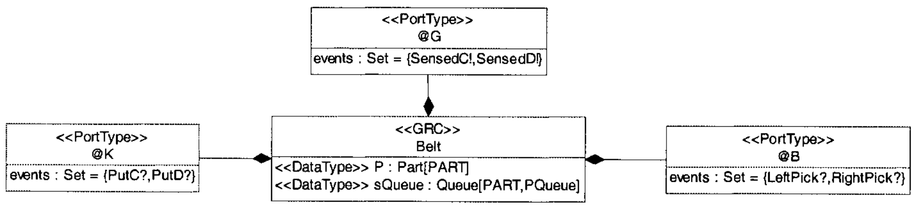
The user uses port @VS to communicate with the belt through its port @K.
The vision system uses port @U to communicate with the belt through its port @G.
The vision system uses port @S to communicate with the robot through its port @C.
The robot uses port @D to communicate with the belt through its port @B and uses port @E to communicate with the tray through its port @W.
The belt has two attributes. One is P that is type of trait Part, and another one is sQueue that is a type of trait Queue.
The vision system has one attribute. It is P that is a type of trait Part.
The robot has five attributes. Three of them are rPrt, lPrt, and P, and are types of trait Part. One of others is inStack of trait Stack, and the last one is inQueue of trait Queue.

The Belt Class

The belt consists of a conveyor belt and a buffer. Previously, the vision system's sensor detected a part and recognized it. The new design incorporates a sensor in the belt, and is located near the vision system. So, the belt will sense when a part has moved into the sensor zone, and inform the vision system. Now, the belt also can accept parts at any time, even when it is in a stop state. The buffer will load parts onto the belt at an even rate whenever the belt is moving. When a part moves into the sensor zone and the belt's sensor detects it, the buffer will stop placing parts onto the belt, but it can still accept parts from the user. The belt can adjust to different speeds between the user and the system accordingly.

```
Class Belt [@K, @G, @B]
Events: PutC?@K, PutD?@K, SensedC!@G, SensedD!@G, LeftPick?@B,
        RightPick?@B
States: *moving, transporting, stop
Attributes: P:PART;sQueue:PQueue
Traits: Part[PART],Queue[PART,PQueue]
Attribute-Function: moving -> {};transporting -> {sQueue};stop
        -> {sQueue};
Transition-Specifications:
     R1: <moving,transporting>; PutC(true); true =>
        sQueue'=append(cup(),sQueue);
     R2: <moving,transporting>; PutD(true); true =>
        sQueue'=append(dish(),sQueue);
     R3: <transporting,stop>; SensedC(true); head(sQueue)=cup()
        => sQueue'=tail(sQueue);
     R4: <transporting,stop>; SensedD(true);
        head(sQueue)=dish() => sQueue'=tail(sQueue);
     R5: <transporting,transporting>; PutC(true); true =>
        sQueue'=append(cup(),sQueue);
     R6: <transporting,transporting>; PutD(true); true =>
        sQueue'=append(dish(),sQueue);
     R7: <stop,transporting>; LeftPick(true); len(sQueue)>0 =>
        true;
     R8: <stop,stop>; PutC(true); true =>
        sQueue'=append(cup(),sQueue);
     R9: <stop,stop>; PutD(true); true =>
        sQueue'=append(dish(),sQueue);
     R10: <stop,moving>; RightPick(true); len(sQueue)=0 =>
        true;
     R11: <stop,transporting>; RightPick(true); len(sQueue)>0
        => true;
     R12: <stop,moving>; LeftPick(true); len(sQueue)=0 => true;
Time-Constraints:
     TCVar1: R1, SensedC, [2, 3], {};
     TCVar3: R7, SensedC, [0, 1], {};
     TCVar5: R11, SensedC, [0, 1], {};
     TCVar2: R2, SensedD, [2, 3], {};
     TCVar4: R7, SensedD, [0, 1], {};
     TCVar6: R11, SensedD, [0, 1], {};
end
```

Figure 86: New Belt TROM class – Class specification



<<PortType>>
@G
events : Set = {SensedC!,SensedD!}
<<PortType>>
@K
events : Set = {PutC?,PutD?}
<<GRC>>
Belt
<<DataType>> P : Part[PART]
<<DataType>> sQueue : Queue[PART,PQueue]
<<PortType>>
@B
events : Set = {LeftPick?,RightPick?}

Figure 87: New Belt TROM class – Class diagram

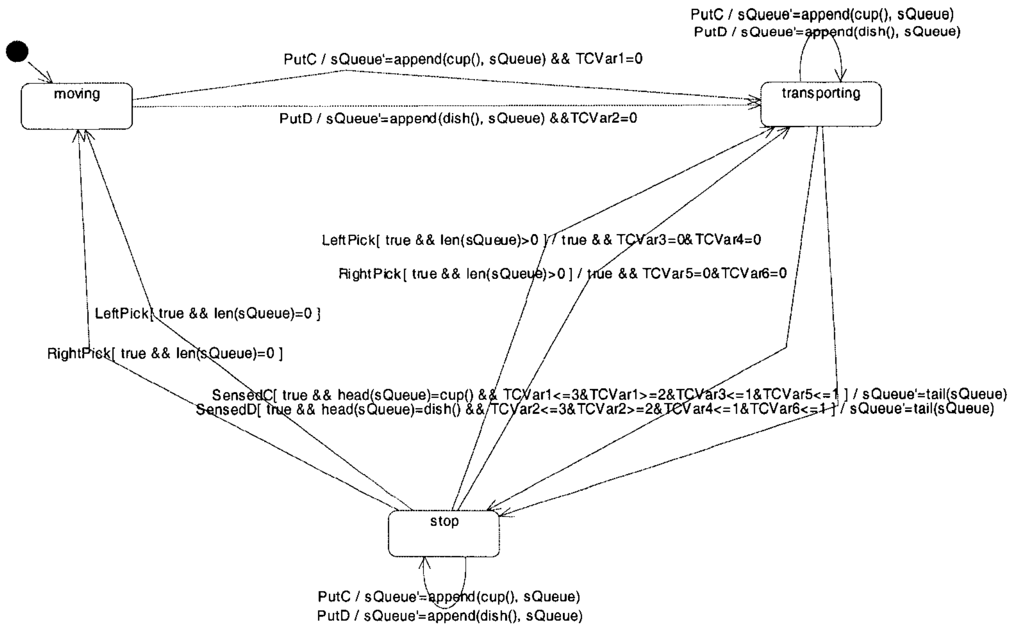

PutC / sQueue'=append(cup(), sQueue)
PutD / sQueue'=append(dish(), sQueue)
PutC / sQueue'=append(cup(), sQueue) && TCVar1=0
moving
transporting
PutD / sQueue'=append(dish(), sQueue) &&TCVar2=0
LeftPick[true && len(sQueue)>0] / true && TCVar3=0&TCVar4=0
RightPick[true && len(sQueue)>0] / true && TCVar5=0&TCVar6=0
LeftPick[true && len(sQueue)=0]
RightPick[true && len(sQueue)=0]
SensedC[true && head(sQueue)=cup() && TCVar1<=3&TCVar1>=2&TCVar3<=1&TCVar5<=1] / sQueue'=tail(sQueue)
SensedD[true && head(sQueue)=dish() && TCVar2<=3&TCVar2>=2&TCVar4<=1&TCVar6<=1] / sQueue'=tail(sQueue)
stop
PutC / sQueue'=append(cup(), sQueue)
PutD / sQueue'=append(dish(), sQueue)

Figure 88: New Belt TROM class – State chart diagram

New Time Constraints for The Belt

(1) The belt will sense the part within 2 time units after the user has placed it onto the belt.

(2) The next part moves into the sensor zone within 1 time unit after the previous one is picked up by the robot.

The first time constraint means that a part will move into the sensor zone in 2 time units if both the convey belt and the buffer are empty. The second time constraint means that if the conveyor belt or the buffer contains more than one part, the next part will move into the sensor zone in 1 time unit after the previous one was picked up by the robot. This mechanism guarantees that parts can be handled continuously.

The Vision System Class

Because the vision system will only handle one part at any time, we removed the Queue data structure from it.

```
Class VisionSystem [@S, @U]
Events: SensedC?@U, SensedD?@U, RecC!@S, RecD!@S
States: *idle, identify
Attributes: P:PART
Traits: Part[PART]
Attribute-Function: idle -> {};identify -> {};
Transition-Specifications:
     R1: <idle,identify>; SensedC(true); true => true;
     R2: <idle,identify>; SensedD(true); true => true;
     R3: <identify,idle>; RecC(true); true => true;
     R4: <identify,idle>; RecD(true); true => true;
```

```
Time-Constraints:
     TCVar1: R1, RecC, [0, 5], {};
     TCVar2: R2, RecD, [0, 5], {};
end
```

Figure 89: New Vision System TROM class – Class Specification

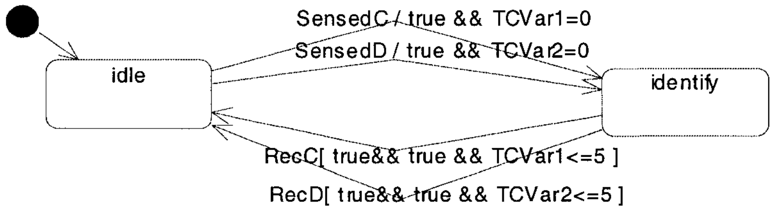

SensedC / true && TCVar1=0
SensedD / true && TCVar2=0
idle
identify
RecC[true&& true && TCVar1<=5]
RecD[true&& true && TCVar2<=5]

Figure 90: New Vision System TROM class – State chart diagram

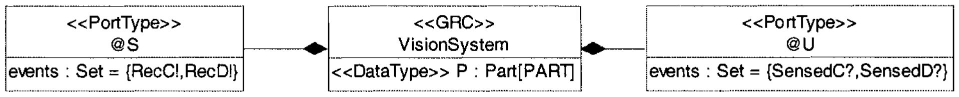

<<PortType>>
@S
events : Set = {RecC!,RecD!}
<<GRC>>
VisionSystem
<<DataType>> P : Part[PART]
<<PortType>>
@U
events : Set = {SensedC?,SensedD?}

Figure 91: New Vision System TROM – class diagram

The Robot Class

The belt will resume moving after the robot has picked up a part. The vision system may send another recognizing event when it performs placing actions. So, we added a queue into the robot to store this kind of event information and add some more transitions into states, S4, S5, S6, and S7 to describe this situation.

```
Class Robot [@D, @C, @E]
Events: RecC?@C, RecD?@C, LeftPick!@D, RightPick!@D, Insert,
        LeftPlace!@E, RightPlace!@E, LPopStack, FreeRight
States: *S1, S2, S3, S4, S5, S6
Attributes: P:PART; rPrt:PART; lPrt:PART; inStack:PStack;
        inQueue:PQueue
Traits: Part[PART],Stack[PART,PStack],Queue[PART,PQueue]
Attribute-Function: S1 -> {inQueue, rPrt};S2 -> {lPrt, inQueue,
        rPrt};S3 -> {rPrt, inQueue};S4 -> {lPrt, inQueue};S5 ->
        {lPrt, inStack, inQueue};S6 -> {inStack};
Transition-Specifications:
     R1: <S1,S1>; RecC(true); true =>
        inQueue'=append(cup(),inQueue);
     R2: <S1,S1>; RecD(true); true =>
        inQueue'=append(dish(),inQueue);
     R3: <S1,S2>; LeftPick(true); len(inQueue)>0 =>
        lPrt'=head(inQueue)&inQueue'=tail(inQueue);
     R4: <S2,S3>; RightPick(true); len(inQueue)>0 =>
        rPrt'=head(inQueue)&inQueue'=tail(inQueue);
     R5: <S2,S2>; RecC(true); true =>
        inQueue'=append(cup(),inQueue);
        R6: <S2,S2>; RecD(true); true =>
           inQueue'=append(dish(),inQueue);
        R7: <S3,S6>; Insert(true); rPrt=lPrt =>
           inStack'=push(rPrt,inStack);
        R8: <S3,S4>; LeftPlace(true); !(lPrt=rPrt) =>
           lPrt'=nullpart();
        R9: <S3,S3>; RecC(true); true =>
           inQueue'=append(cup(),inQueue);
        R10: <S3,S3>; RecD(true); true =>
           inQueue'=append(dish(),inQueue);
        R11: <S4,S1>; RightPlace(true); isEmpty(inStack) =>
           rPrt'=nullpart();
        R12: <S4,S5>; LPopStack(true); !(isEmpty(inStack)) =>
           lPrt'=top(inStack)&inStack'=pop(inStack);
        R13: <S4,S4>; RecC(true); true =>
           inQueue'=append(cup(),inQueue);
        R14: <S4,S4>; RecD(true); true =>
           inQueue'=append(dish(),inQueue);
        R15: <S5,S2>; RightPlace(true); true => rPrt'=nullpart();
        R16: <S5,S5>; RecC(true); true =>
           inQueue'=append(cup(),inQueue);
        R17: <S5,S5>; RecD(true); true =>
           inQueue'=append(dish(),inQueue);
        R18: <S6,S2>; FreeRight(true); true => rPrt'=nullpart();
   Time-Constraints:
        TCVar1: R1, LeftPick, [0, 2], {};
        TCVar2: R2, LeftPick, [0, 2], {};
        TCVar9: R11, LeftPick, [0, 1], {};
        TCVar3: R6, RightPick, [0, 2], {};
        TCVar4: R5, RightPick, [0, 2], {};
        TCVar8: R15, RightPick, [0, 1], {};
        TCVar7: R4, LeftPlace, [2, 3], {};
        TCVar6: R4, RightPlace, [3, 4], {};
        TCVar5: R4, RightPlace, [3, 4], {};
   end
```

Figure 92: New Robot TROM class – Class specification

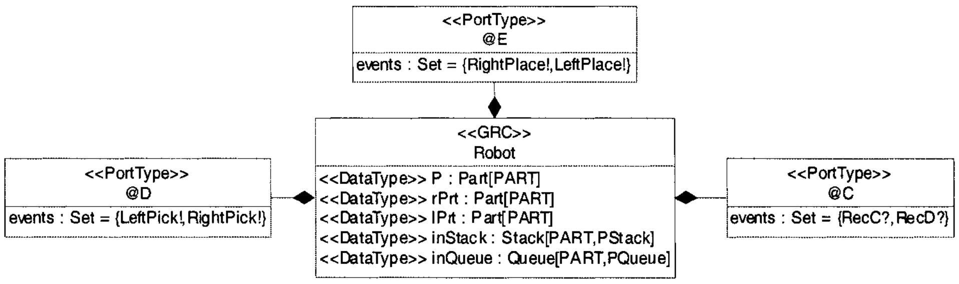
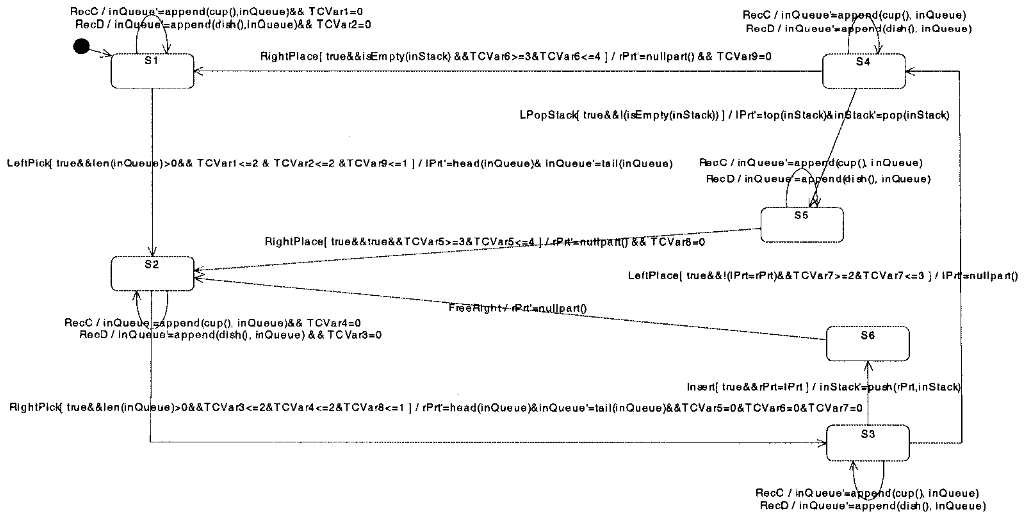

<<PortType>>
@E
events : Set = {RightPlace!,LeftPlace!}
<<GRC>>
Robot
<<DataType>> P : Part[PART]
<<DataType>> rPrt : Part[PART]
<<DataType>> lPrt : Part[PART]
<<DataType>> inStack : Stack[PART,PStack]
<<DataType>> inQueue : Queue[PART,PQueue]
<<PortType>>
@D
events : Set = {LeftPick!,RightPick!}
<<PortType>>
@C
events : Set = {RecC?,RecD?}

Figure 93: New Robot TROM class – Class diagram


RecC / inQueue'=append(cup(),inQueue)&& TCVar1=0
RecD / inQueue'=append(dish(),inQueue)&& TCVar2=0
S1
RightPlace[true&&isEmpty(inStack) &&TCVar6>=3&TCVar6<=4] / rPrt'=nullpart() && TCVar9=0
RecC / inQueue'=append(cup(), inQueue)
RecD / inQueue'=append(dish(), inQueue)
S4
LPopStack[true&&!(isEmpty(inStack))] / lPrt'=top(inStack)&inStack'=pop(inStack)
LeftPick[true&&len(inQueue)>0&& TCVar1<=2 & TCVar2<=2 &TCVar9<=1] / lPrt'=head(inQueue)& inQueue'=tail(inQueue)
RecC / inQueue'=append(cup(), inQueue)
RecD / inQueue'=append(dish(), inQueue)
S5
RightPlace[true&&true&&TCVar5>=3&TCVar5<=4] / rPrt'=nullpart() && TCVar8=0
S2
LeftPlace[true&&!(lPrt=rPrt)&&TCVar7>=2&TCVar7<=3] / lPrt'=nullpart()
FreeRight / rPrt'=nullpart()
RecC / inQueue'=append(cup(), inQueue)&& TCVar4=0
RecD / inQueue'=append(dish(), inQueue) && TCVar3=0
S6
Insert[true&&rPrt=lPrt] / inStack'=push(rPrt,inStack)
RightPick[true&&len(inQueue)>0&&TCVar3<=2&TCVar4<=2&TCVar8<=1] / rPrt'=head(inQueue)&inQueue'=tail(inQueue)&&TCVar5=0&TCVar6=0&TCVar7=0
S3
RecC / inQueue'=append(cup(), inQueue)
RecD / inQueue'=append(dish(), inQueue)

```
S1 - Both arms are free
S2 - Left arm is not free, right arm is free
S3 - Right arm is not free, left arm is not free
S4 - Left arm is free, right arm is not free
S5 - Left arm removing from Stack and not free, right arm is
     not free
S6 - Right arm is inserting into Stack, left arm is not free
```

Figure 94: New Robot TROM class – State chart diagram

The Subsystem Configuration Specification (SCS)

Because the interactions among the user, belt, and vision system have been changed, the collaboration diagram should be changed too.

```
SCS BasicRobot
     Includes:
     Instantiate:
          U1::User[@VS:1];
          V1::VisionSystem[@S:1, @U:1];
          R1::Robot[@D:1, @C:1, @E:1];
          T1::Tray[@W:1];
          B1::Belt[@K:1, @G:1, @B:1];
     Configure:
          B1.@K1:@K <-> U1.@VS1:@VS;
          B1.@G1:@G <-> V1.@U1:@U;
          R1.@C1:@C <-> V1.@S1:@S;
          R1.@E1:@E <-> T1.@W1:@W;
          B1.@B1:@B <-> R1.@D1:@D;
end
```

Figure 95: SCS – New specification

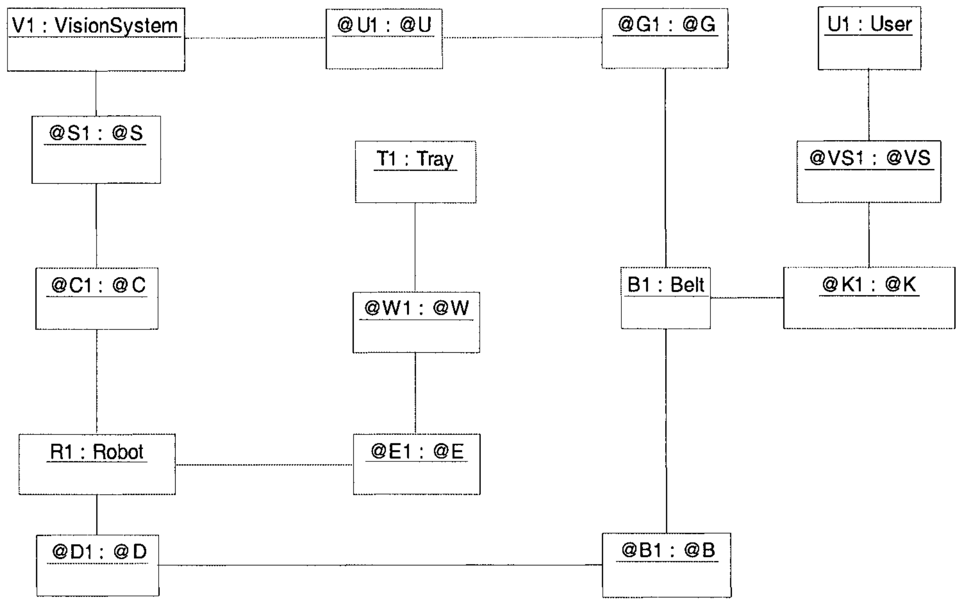
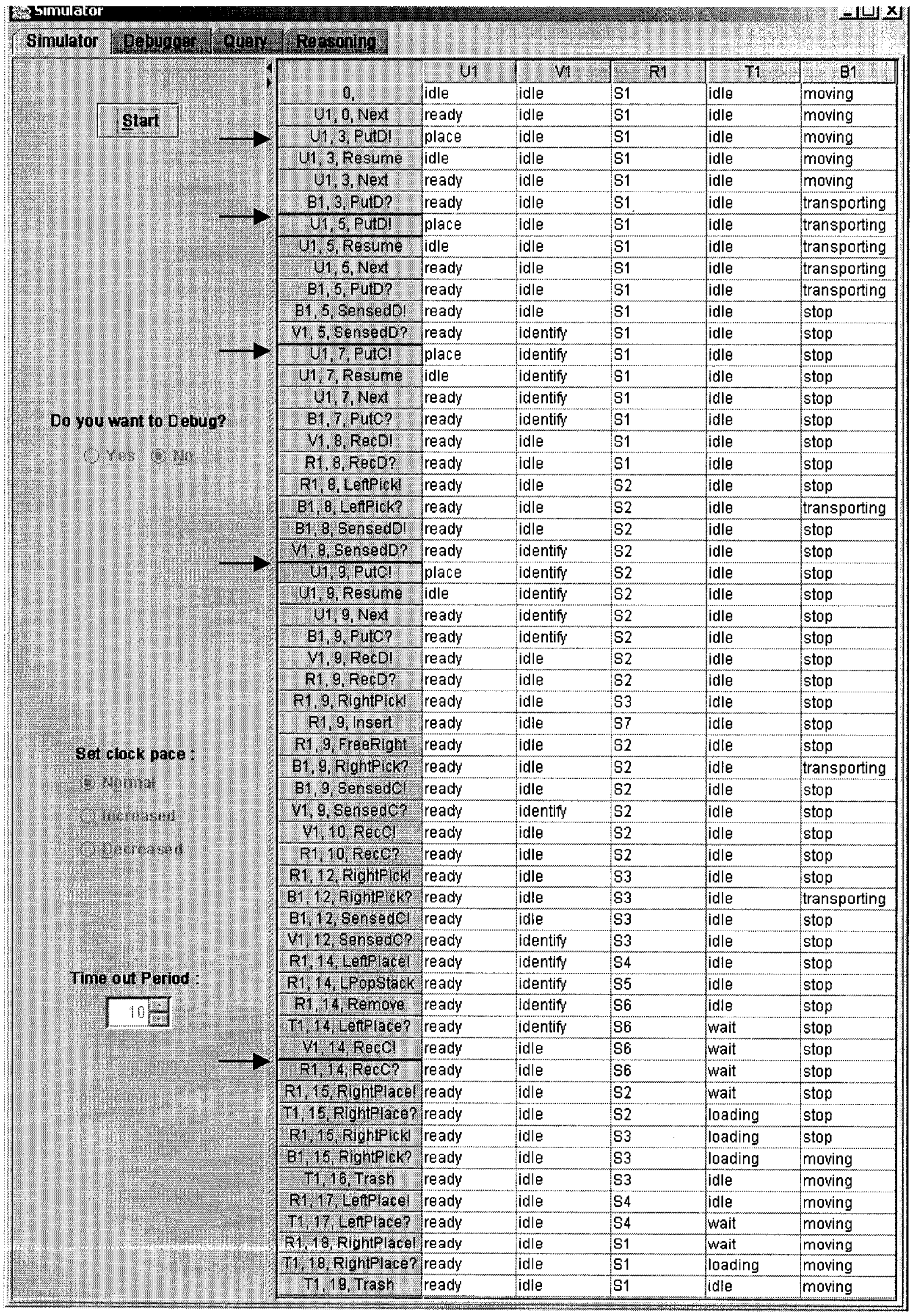


Figure 96: SCS – New Collaboration diagram

The Simulation Result

Figure 97 displays the simulation results. It shows that the assembly was successful. Regardless of which state the belt is in, it still can accept parts that the user places on it. For example, the first four putting events marked by black arrows are triggered in the moving, transporting, and stop states of the belt respectively. Also, the robot can store messages when it performs placing actions. For instance, the event marked by last black arrow shows that the robot received eventRecC while the robot was in state S6 where it was in the process of placing the part held in the right arm onto the tray.



Do you want to Debug?

Yes [✓ No]

Set clock pace :

[✓ Normal]

Increased

Decreased

Time out Period : 10

	U1	V1	R1	T1	B1
0,	idle	idle	S1	idle	moving
U1, 0, Next	ready	idle	S1	idle	moving
U1, 3, PutD!	place	idle	S1	idle	moving
U1, 3, Resume	idle	idle	S1	idle	moving
U1, 3, Next	ready	idle	S1	idle	moving
B1, 3, PutD?	ready	idle	S1	idle	transporting
U1, 5, PutD!	place	idle	S1	idle	transporting
U1, 5, Resume	idle	idle	S1	idle	transporting
U1, 5, Next	ready	idle	S1	idle	transporting
B1, 5, PutD?	ready	idle	S1	idle	transporting
B1, 5, SensedD!	ready	idle	S1	idle	stop
V1, 5, SensedD?	ready	identify	S1	idle	stop
U1, 7, PutC!	place	identify	S1	idle	stop
U1, 7, Resume	idle	identify	S1	idle	stop
U1, 7, Next	ready	identify	S1	idle	stop
B1, 7, PutC?	ready	identify	S1	idle	stop
V1, 8, RecD!	ready	idle	S1	idle	stop
R1, 8, RecD?	ready	idle	S1	idle	stop
R1, 8, LeftPick!	ready	idle	S2	idle	stop
B1, 8, LeftPick?	ready	idle	S2	idle	transporting
B1, 8, SensedD!	ready	idle	S2	idle	stop
V1, 8, SensedD?	ready	identify	S2	idle	stop
U1, 9, PutC!	place	identify	S2	idle	stop
U1, 9, Resume	idle	identify	S2	idle	stop
U1, 9, Next	ready	identify	S2	idle	stop
B1, 9, PutC?	ready	identify	S2	idle	stop
V1, 9, RecD!	ready	idle	S2	idle	stop
R1, 9, RecD?	ready	idle	S2	idle	stop
R1, 9, RightPick!	ready	idle	S3	idle	stop
R1, 9, Insert	ready	idle	S7	idle	stop
R1, 9, FreeRight	ready	idle	S2	idle	stop
B1, 9, RightPick?	ready	idle	S2	idle	transporting
B1, 9, SensedC!	ready	idle	S2	idle	stop
V1, 9, SensedC?	ready	identify	S2	idle	stop
V1, 10, RecC!	ready	idle	S2	idle	stop
R1, 10, RecC?	ready	idle	S2	idle	stop
R1, 12, RightPick!	ready	idle	S3	idle	stop
B1, 12, RightPick?	ready	idle	S3	idle	transporting
B1, 12, SensedC!	ready	idle	S3	idle	stop
V1, 12, SensedC?	ready	identify	S3	idle	stop
R1, 14, LeftPlace!	ready	identify	S4	idle	stop
R1, 14, LPopStack	ready	identify	S5	idle	stop
R1, 14, Remove	ready	identify	S6	idle	stop
T1, 14, LeftPlace?	ready	identify	S6	wait	stop
V1, 14, RecC!	ready	idle	S6	wait	stop
R1, 14, RecC?	ready	idle	S6	wait	stop
R1, 15, RightPlace!	ready	idle	S2	wait	stop
T1, 15, RightPlace?	ready	idle	S2	loading	stop
R1, 15, RightPick!	ready	idle	S3	loading	stop
B1, 15, RightPick?	ready	idle	S3	loading	moving
T1, 16, Trash	ready	idle	S3	idle	moving
R1, 17, LeftPlace!	ready	idle	S4	idle	moving
T1, 17, LeftPlace?	ready	idle	S4	wait	moving
R1, 18, RightPlace!	ready	idle	S1	wait	moving
T1, 18, RightPlace?	ready	idle	S1	loading	moving
T1, 19, Trash	ready	idle	S1	idle	moving

Figure 97: Simulation Result of Buffered Belt Modeling

7.5 Remodeling with Parameterized Events

In this section, we will use the modified versions of the self-controlled user and buffered belt as prototypes to remodel the system with parameterized events. The remodeling also demonstrates that the simulator is ready to acceptparameterized events.

7.5.1 The LSL Trait – Part

For remodeling the system with parameterized events, we defined the LSL trait as follows:

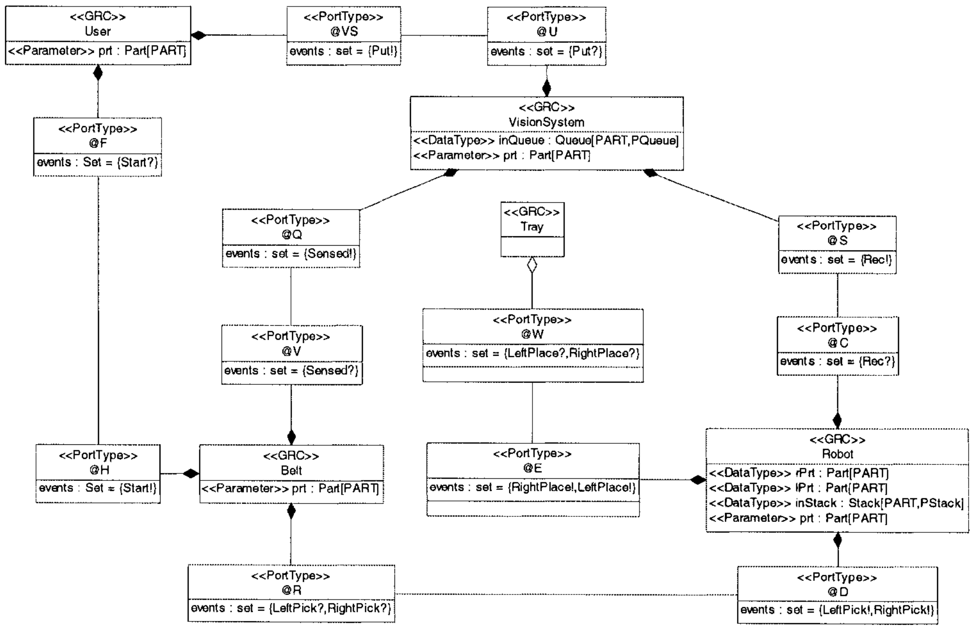
```
Trait: Part(P)
    Includes: Boolean
    Introduce:
            cup      :   -> P;
            dish     :   -> P;
            nullpart :   -> P;
end
```

Figure 98: The Part trait

7.5.2 Self-controlled User Modeling with Parameterized Events

Class Diagram for Robotics System

All classes except class tray have a parameter prt for storing the part information. The number of events decreases too.



<<GRC>>
User
<<Parameter>> prt : Part[PART]
<<PortType>>
@VS
events : set = {Put!}
<<PortType>>
@U
events : set = {Put?}
<<PortType>>
@F
events : Set = {Start?}
<<GRC>>
VisionSystem
<<DataType>> inQueue : Queue[PART,PQueue]
<<Parameter>> prt : Part[PART]
<<PortType>>
@Q
events : set = {Sensed!}
<<GRC>>
Tray
<<PortType>>
@S
events : set = {Rec!}
<<PortType>>
@V
events : set = {Sensed?}
<<PortType>>
@W
events : set = {LeftPlace?,RightPlace?}
<<PortType>>
@C
events : set = {Rec?}
<<PortType>>
@H
events : Set = {Start!}
<<GRC>>
Belt
<<Parameter>> prt : Part[PART]
<<PortType>>
@E
events : set = {RightPlace!,LeftPlace!}
<<GRC>>
Robot
<<DataType>> rPrt : Part[PART]
<<DataType>> lPrt : Part[PART]
<<DataType>> inStack : Stack[PART,PStack]
<<Parameter>> prt : Part[PART]
<<PortType>>
@R
events : set = {LeftPick?,RightPick?}
<<PortType>>
@D
events : set = {LeftPick!,RightPick!}

Figure 99: Robotics System class Diagram

The User Class with Parameterized Events

The user class has a parameter prt to represent the types of parts that are placed onto the conveyor belt. This information about the part type is passed to the belt through the argument of the event Put.

```
Class User [@VS, @F]
Events: Next, Put!@VS, Start?@F, Resume
States: *idle, ready, place, wait
Attributes: prt:PART
Traits: Part[PART]
Attribute-Function: idle -> {};ready -> {prt};place -> {};wait
          -> {};
Parameter-Specifications:
     Put: prt;
Transition-Specifications:
     R1: <idle,ready>; Next[](true); true => true;
     R2: <ready,place>; Put[prt](true); true => true;
     R3: <place,wait>; Start[](true); true => true;
     R4: <wait,idle>; Resume[](true); true => true;
Time-Constraints:

end
```

Figure 100: User TROM class – class specification

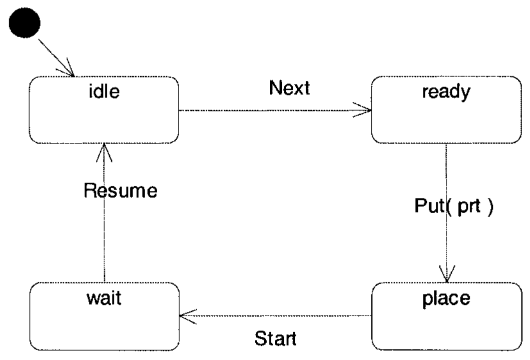


Figure 101: User TROM class – state chart diagram

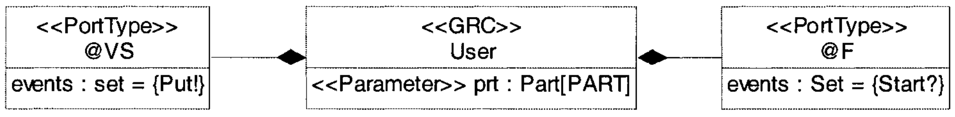


Figure 102: User TROM class – class diagram

The Belt Class with Parameterized Events

The belt has a parameter prt, but it is not used in the transitions. All events having parameters are input events. They just receive parameters from the output events of other TROM objects.

```
Class Belt [@V, @R, @H]
Events: Sensed?@V, RightPick?@R, LeftPick?@R, Start!@H
States: *active, stop, toActive
Attributes: prt:PART
Traits: Part[PART]
Attribute-Function: active -> {prt};stop -> {prt};toActive ->
          {};
Parameter-Specifications:
     LeftPick: prt;
     RightPick: prt;
     Sensed: prt;
Transition-Specifications:
     R1: <active,stop>; Sensed[prt](true); true => true;
     R2: <stop,toActive>; RightPick[prt](true); true => true;
     R3: <stop,toActive>; LeftPick[prt](true); true => true;
     R4: <toActive,active>; Start[](true); true => true;
Time-Constraints:
     TCVar1: R2, Start, [0, 1], {};
     TCVar2: R3, Start, [0, 1], {};
end
```

Figure 103: Belt TROM class – class specification

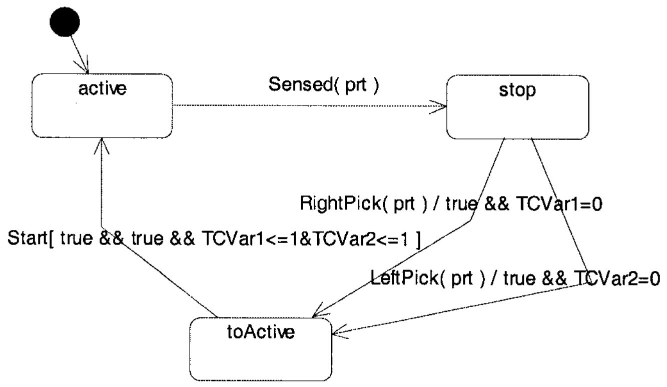

active
Sensed(prt)
stop
RightPick(prt) / true && TCVar1=0
Start[true && true && TCVar1<=1&TCVar2<=1]
LeftPick(prt) / true && TCVar2=0
toActive

Figure 104: Belt TROM class – state chart diagram

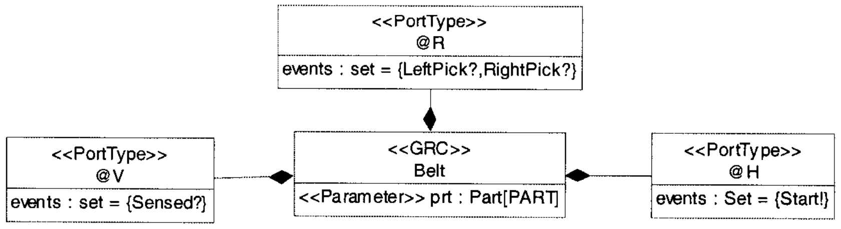


Figure 105: Belt TROM class – class diagram

The Vision System Class with Parameterized Events

The vision system stores the parameter *prt* into the queue *inQueue*. Later on, it recognizes parts from the elements in the queue.

```
Class VisionSystem [@U, @Q, @S]
Events: Put?@U, Sensed!@Q, Rec!@S
States: *Monitor, active, identify
Attributes: inQueue:PQueue;prt:PART
Traits: Queue[PART,PQueue],Part[PART]
Attribute-Function: Monitor -> {inQueue, prt};active ->
          {inQueue, prt};identify -> {inQueue, prt};
Parameter-Specifications:
     Put: prt;
     Rec: prt;
     Sensed: prt;
Transition-Specifications:
     R1: <Monitor,active>; Put[prt](true); true =>
          inQueue'=append(prt,inQueue);
     R2: <active,active>; Put[prt](true); true =>
          inQueue'=append(prt,inQueue);
     R3: <active,identify>; Sensed[prt=head(inQueue)](true);
          true => true;
     R4: <identify,active>; Rec[prt](true); len(inQueue)>1 =>
          inQueue'=tail(inQueue);
     R5: <identify,Monitor>; Rec[prt](true); len(inQueue)=1 =>
          inQueue'=tail(inQueue);
     R6: <identify,identify>; Put[prt](true); true =>
          inQueue'=append(prt,inQueue);
Time-Constraints:
     TCVar1: R1, Sensed, [0, 2], {};
     TCVar4: R4, Sensed, [0, 2], {};
     TCVar2: R3, Rec, [4, 5], {};
     TCVar3: R3, Rec, [4, 5], {};
end
```

Figure 106: Vision System TROM class – class specification

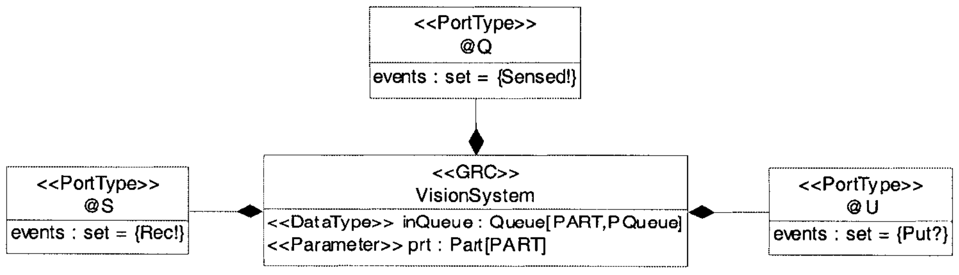


Figure 107: Vision System TROM class – class diagram

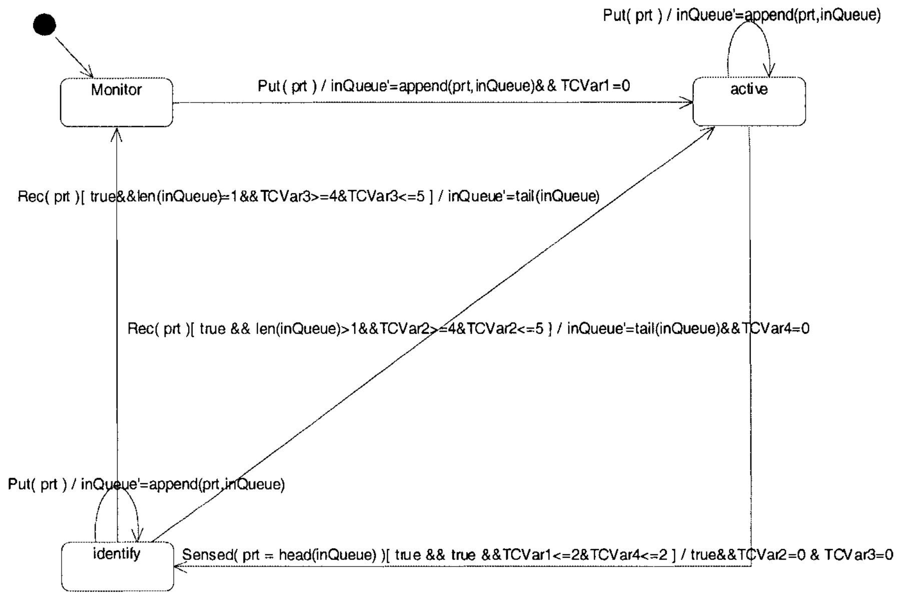

Put(prt) / inQueue'=append(prt,inQueue)
Monitor
Put(prt) / inQueue'=append(prt,inQueue)&& TCVar1=0
active
Rec(prt)[true&&len(inQueue)=1&&TCVar3>=4&TCVar3<=5] / inQueue'=tail(inQueue)
Rec(prt)[true && len(inQueue)>1&&TCVar2>=4&TCVar2<=5] / inQueue'=tail(inQueue)&&TCVar4=0
Put(prt) / inQueue'=append(prt,inQueue)
identify
Sensed(prt = head(inQueue))[true && true &&TCVar1<=2&TCVar4<=2] / true&&TCVar2=0 & TCVar3=0

Figure 108: Vision System TROM class – state chart diagram

The Robot Class with Parameterized Events

The robot class waits for any recognized parts and saves this information carried by the parameter into a queue for later identification.

```
Class Robot [@D, @C, @E]
Events: Rec?@C, LeftPick!@D, RightPick!@D, Insert,
          LeftPlace!@E, RightPlace!@E, LPopStack, FreeRight,
          Remove
States: *S1, S2, S3, S4, S5, S6, S7, S8, S9
Attributes: rPrt:PART;lPrt:PART;inStack:PStack;prt:PART
Traits: Part[PART],Stack[PART,PStack]
Attribute-Function: S1 -> {rPrt, prt};S2 -> {prt};S3 -> {lPrt,
          rPrt, prt};S4 -> {prt};S5 -> {rPrt};S6 -> {lPrt};S7 -
          > {inStack};S8 -> {inStack};S9 -> {lPrt};
Parameter-Specifications:
     LeftPick: prt;
     Rec: prt;
     RightPick: prt;
Transition-Specifications:
     R1: <S1,S2>; Rec[prt](true); true => true;
     R2: <S2,S3>; LeftPick[prt](true); true => lPrt'=prt;
     R3: <S3,S4>; Rec[prt](true); true => true;
     R4: <S4,S5>; RightPick[prt](true); true => rPrt'=prt;
     R5: <S5,S8>; Insert[](true); rPrt=lPrt =>
          inStack'=push(rPrt,inStack);
     R6: <S5,S6>; LeftPlace[](true); !(lPrt=rPrt) =>
          lPrt'=nullpart();
     R7: <S6,S1>; RightPlace[](true); isEmpty(inStack) =>
          rPrt'=nullpart();
     R8: <S6,S9>; LPopStack[](true); !(isEmpty(inStack)) =>
          lPrt'=top(inStack);
     R9: <S7,S3>; RightPlace[](true); true => rPrt'=nullpart();
     R10: <S8,S3>; FreeRight[](true); true => rPrt'=nullpart();
     R11: <S9,S7>; Remove[](true); true =>
          inStack'=pop(inStack);
Time-Constraints:
     TCVar1: R1, LeftPick, [0, 2], {};
     TCVar2: R3, RightPick, [0, 2], {};
     TCVar3: R4, LeftPlace, [0, 1], {};
     TCVar4: R4, RightPlace, [1, 2], {};
     TCVar5: R4, RightPlace, [1, 2], {};
end
```

Figure 109: Robot TROM class – class specification

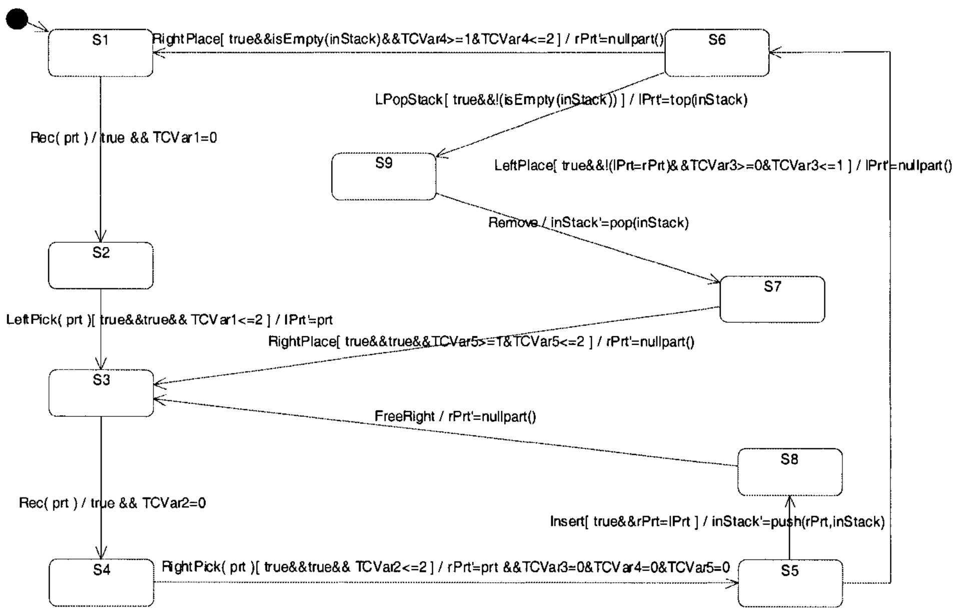

S1
RightPlace[true&&isEmpty(inStack)&&TCVar4>=1&TCVar4<=2] / rPrt'=nullpart()
S6
LPopStack[true&&!(isEmpty(inStack))] / lPrt'=top(inStack)
Rec(prt) / true && TCVar1=0
S9
LeftPlace[true&&!(lPrt=rPrt)&&TCVar3>=0&TCVar3<=1] / lPrt'=nullpart()
Remove / inStack'=pop(inStack)
S2
S7
LeftPick(prt)[true&&true&& TCVar1<=2] / lPrt'=prt
RightPlace[true&&true&&TCVar5>=1&TCVar5<=2] / rPrt'=nullpart()
S3
FreeRight / rPrt'=nullpart()
S8
Rec(prt) / true && TCVar2=0
Insert[true&&rPrt=lPrt] / inStack'=push(rPrt,inStack)
S4
RightPick(prt)[true&&true&& TCVar2<=2] / rPrt'=prt &&TCVar3=0&TCVar4=0&TCVar5=0
S5

Figure 110: Robot TROM class – state chart diagram

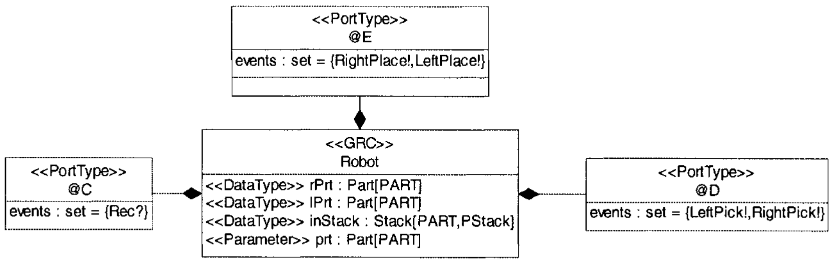

<<PortType>>
@E
events : set = {RightPlace!,LeftPlace!}
<<GRC>>
Robot
<<DataType>> rPrt : Part[PART]
<<DataType>> lPrt : Part[PART]
<<DataType>> inStack : Stack[PART,PStack]
<<Parameter>> prt : Part[PART]
<<PortType>>
@C
events : set = {Rec?}
<<PortType>>
@D
events : set = {LeftPick!,RightPick!}

Figure 111: Robot TROM class – class diagram

The Tray Class with Parameterized Events

The tray class does not define parameters.

```
Class Tray [@W]
Events: LeftPlace?@W, RightPlace?@W, Trash
States: *idle, wait, loading
Attributes:
```

```
Traits:
Attribute-Function: idle -> {};wait -> {};loading -> {};
Parameter-Specifications:

Transition-Specifications:
     R1: <idle,wait>; LeftPlace[](true); true => true;
     R2: <wait,loading>; RightPlace[](true); true => true;
     R3: <loading,idle>; Trash[](true); true => true;
Time-Constraints:
     TCVar1: R2, Trash, [1, 2], {};
end
```

Figure 112: Tray TROM class – class specification

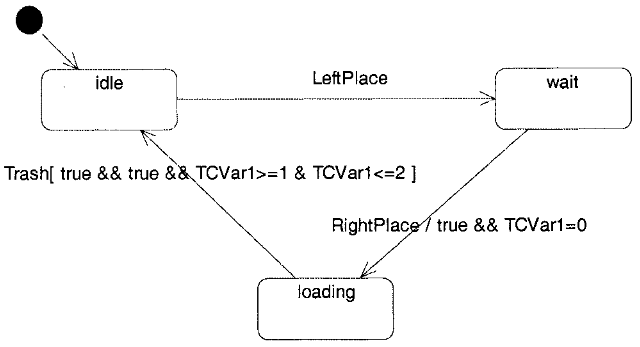

idle
LeftPlace
wait
Trash[true && true && TCVar1>=1 & TCVar1<=2]
RightPlace / true && TCVar1=0
loading

Figure 113: Tray TROM class – state chart specification

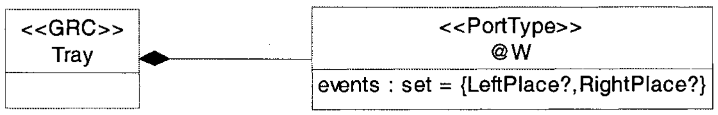


Figure 114: Tray TROM class – class specification

The Sample Simulation Event List

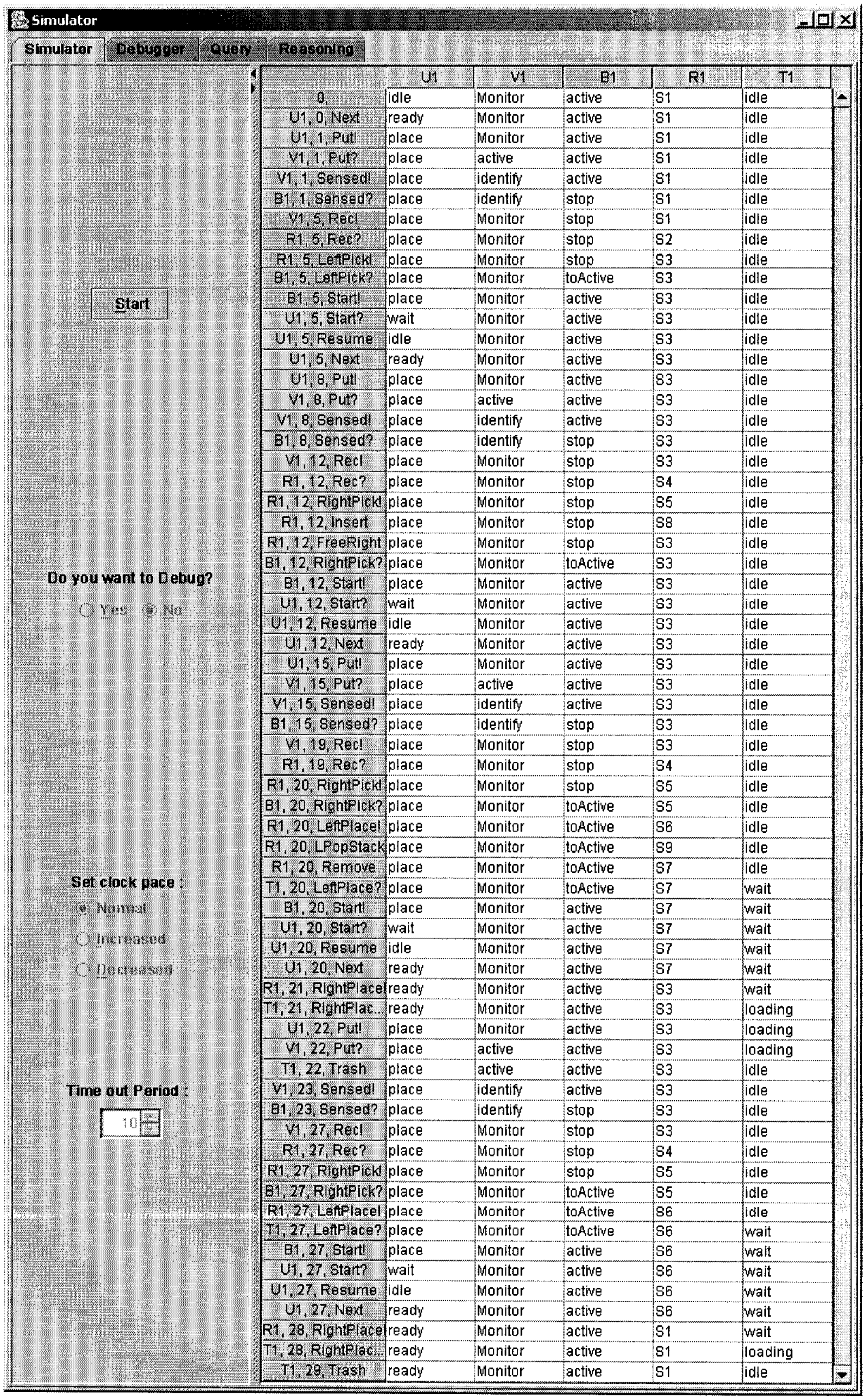
The Simulation Event List simulates the same information as the above models to the system except that the part identification is passed by parameter expression instead of the event itself.

```
SEL: BasicRobot
      U1, Put[prt=dish()], @VS1, 1;
      U1, Put[prt=dish()], @VS1, 8;
      U1, Put[prt=cup()], @VS1, 15;
      U1, Put[prt=cup()], @VS1, 22;
end
```

Figure 115: Sample Simulation Event List

The Simulation Result of Self-controlled User Modeling

Figure 116 displays the successful simulation results of the self-controlled user model.



	U1	V1	B1	R1	T1
0,	idle	Monitor	active	S1	idle
U1, 0, Next	ready	Monitor	active	S1	idle
U1, 1, Put!	place	Monitor	active	S1	idle
V1, 1, Put?	place	active	active	S1	idle
V1, 1, Sensed!	place	identify	active	S1	idle
B1, 1, Sensed?	place	identify	stop	S1	idle
V1, 5, Rec!	place	Monitor	stop	S1	idle
R1, 5, Rec?	place	Monitor	stop	S2	idle
R1, 5, LeftPick!	place	Monitor	stop	S3	idle
B1, 5, LeftPick?	place	Monitor	toActive	S3	idle
B1, 5, Start!	place	Monitor	active	S3	idle
U1, 5, Start?	wait	Monitor	active	S3	idle
U1, 5, Resume	idle	Monitor	active	S3	idle
U1, 5, Next	ready	Monitor	active	S3	idle
U1, 8, Put!	place	Monitor	active	S3	idle
V1, 8, Put?	place	active	active	S3	idle
V1, 8, Sensed!	place	identify	active	S3	idle
B1, 8, Sensed?	place	identify	stop	S3	idle
V1, 12, Rec!	place	Monitor	stop	S3	idle
R1, 12, Rec?	place	Monitor	stop	S4	idle
R1, 12, RightPick!	place	Monitor	stop	S5	idle
R1, 12, Insert	place	Monitor	stop	S8	idle
R1, 12, FreeRight	place	Monitor	stop	S3	idle
B1, 12, RightPick?	place	Monitor	toActive	S3	idle
B1, 12, Start!	place	Monitor	active	S3	idle
U1, 12, Start?	wait	Monitor	active	S3	idle
U1, 12, Resume	idle	Monitor	active	S3	idle
U1, 12, Next	ready	Monitor	active	S3	idle
U1, 15, Put!	place	Monitor	active	S3	idle
V1, 15, Put?	place	active	active	S3	idle
V1, 15, Sensed!	place	identify	active	S3	idle
B1, 15, Sensed?	place	identify	stop	S3	idle
V1, 19, Rec!	place	Monitor	stop	S3	idle
R1, 19, Rec?	place	Monitor	stop	S4	idle
R1, 20, RightPick!	place	Monitor	stop	S5	idle
B1, 20, RightPick?	place	Monitor	toActive	S5	idle
R1, 20, LeftPlace!	place	Monitor	toActive	S6	idle
R1, 20, LPopStack	place	Monitor	toActive	S9	idle
R1, 20, Remove	place	Monitor	toActive	S7	idle
T1, 20, LeftPlace?	place	Monitor	toActive	S7	wait
B1, 20, Start!	place	Monitor	active	S7	wait
U1, 20, Start?	wait	Monitor	active	S7	wait
U1, 20, Resume	idle	Monitor	active	S7	wait
U1, 20, Next	ready	Monitor	active	S7	wait
R1, 21, RightPlace!	ready	Monitor	active	S3	wait
T1, 21, RightPlac...	ready	Monitor	active	S3	loading
U1, 22, Put!	place	Monitor	active	S3	loading
V1, 22, Put?	place	active	active	S3	loading
T1, 22, Trash	place	active	active	S3	idle
V1, 23, Sensed!	place	identify	active	S3	idle
B1, 23, Sensed?	place	identify	stop	S3	idle
V1, 27, Rec!	place	Monitor	stop	S3	idle
R1, 27, Rec?	place	Monitor	stop	S4	idle
R1, 27, RightPick!	place	Monitor	stop	S5	idle
B1, 27, RightPick?	place	Monitor	toActive	S5	idle
R1, 27, LeftPlace!	place	Monitor	toActive	S6	idle
T1, 27, LeftPlace?	place	Monitor	toActive	S6	wait
B1, 27, Start!	place	Monitor	active	S6	wait
U1, 27, Start?	wait	Monitor	active	S6	wait
U1, 27, Resume	idle	Monitor	active	S6	wait
U1, 27, Next	ready	Monitor	active	S6	wait
R1, 28, RightPlace!	ready	Monitor	active	S1	wait
T1, 28, RightPlac...	ready	Monitor	active	S1	loading
T1, 29, Trash	ready	Monitor	active	S1	idle

Figure 116: The Simulation Result of Self-controlled User modeling

7.5.3 Buffered Belt Modeling with Parameterized Events

The User Class with Parameterized Events

The User class has a parameter *prt* to represent the types of parts that are placed on to the conveyor belt. This part types data is passed to the belt through the argument of the event *Put*.

```
Class User [@VS]
Events: Next, Put!@VS, Resume
States: *idle, ready, place
Attributes: prt:PART
Traits: Part[PART]
Attribute-Function: idle -> {};ready -> {prt};place -> {};
Parameter-Specifications:
     Put: prt;
Transition-Specifications:
     R1: <idle,ready>; Next[](true); true => true;
     R2: <ready,place>; Put[prt](true); true => true;
     R3: <place,idle>; Resume[](true); true => true;
Time-Constraints:

end
```

Figure 117: User TROM class – class specification

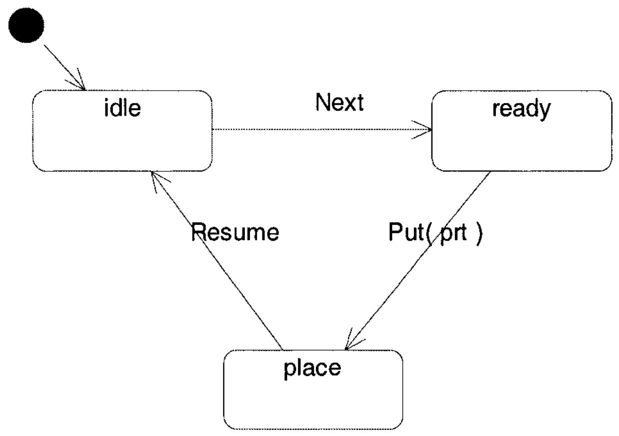


Figure 118: User TROM class – state chart diagram

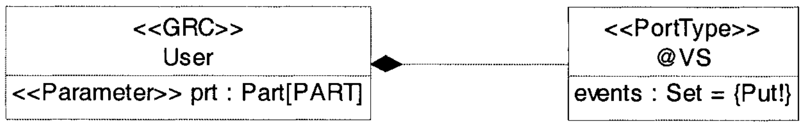


Figure 119: User TROM class – class diagram

The Belt Class with Parameterized Events

Now, the belt class has a parameter *prt* to represent the parts passing on it. The input event *Put* will pass the type of part through the parameter *prt* to the belt from the user class, and then the belt will pass the value to the vision system through the argument of event *Sensed.* The Belt class uses a queue to save the information about parts carried by the parameters.

```
Class Belt [@K, @G, @B]
Events: Put?@K, Sensed!@G, LeftPick?@B, RightPick?@B
States: *moving, transporting, stop
Attributes: sQueue:PQueue;prt:PART
Traits: Queue[PART,PQueue],Part[PART]
Attribute-Function: moving -> {prt};transporting -> {sQueue,
         prt};stop -> {prt, sQueue};
Parameter-Specifications:
     Put: prt;
     Sensed: prt;
Transition-Specifications:
     R1: <moving,transporting>; Put[prt](true); true =>
         sQueue'=append(prt,sQueue);
     R2: <transporting,stop>; Sensed[prt=head(sQueue)](true);
         true => sQueue'=tail(sQueue);
     R3: <transporting,transporting>; Put[prt](true); true =>
         sQueue'=append(prt,sQueue);
     R4: <stop,transporting>; LeftPick[](true); len(sQueue)>0
         => true;
     R5: <stop,stop>; Put[prt](true); true =>
         sQueue'=append(prt,sQueue);
     R6: <stop,moving>; RightPick[](true); len(sQueue)=0 =>
         true;
     R7: <stop,transporting>; RightPick[](true); len(sQueue)>0
         => true;
     R8: <stop,moving>; LeftPick[](true); len(sQueue)=0 =>
         true;
Time-Constraints:
     TCVar1: R1, Sensed, [2, 3], {};
     TCVar3: R4, Sensed, [0, 1], {};
     TCVar4: R7, Sensed, [0, 1], {};
end
```

Figure 120: Belt TROM class – class specification

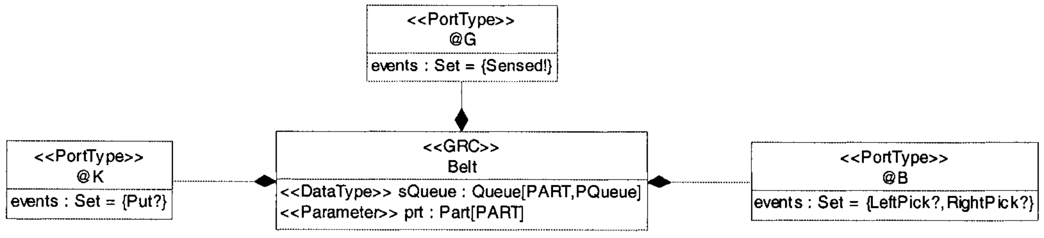

<<PortType>>
@G
events : Set = {Sensed!}
<<PortType>>
@K
events : Set = {Put?}
<<GRC>>
Belt
<<DataType>> sQueue : Queue[PART,PQueue]
<<Parameter>> prt : Part[PART]
<<PortType>>
@B
events : Set = {LeftPick?,RightPick?}

Figure 121: Belt TROM class – class diagram

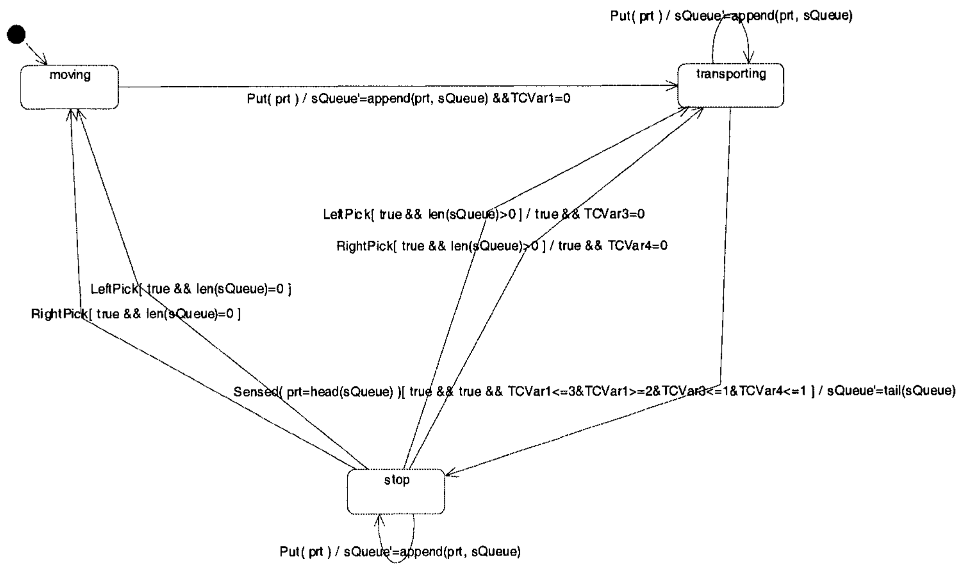

Put(prt) / sQueue'=append(prt, sQueue)
moving
transporting
Put(prt) / sQueue'=append(prt, sQueue) &&TCVar1=0
LeftPick[true && len(sQueue)>0] / true && TCVar3=0
RightPick[true && len(sQueue)>0] / true && TCVar4=0
LeftPick[true && len(sQueue)=0]
RightPick[true && len(sQueue)=0]
Sensed(prt=head(sQueue))[true && true && TCVar1<=3&TCVar1>=2&TCVar3<=1&TCVar4<=1] / sQueue'=tail(sQueue)
stop
Put(prt) / sQueue'=append(prt, sQueue)

Figure 122: Belt TROM class – state chart diagram

The Vision System Class with Parameterized Events

After the vision system recognizes the part, it will transfer the part information to the robot.

```
Class VisionSystem [@S, @U]
Events: Sensed?@U, Rec!@S
States: *idle, identify
Attributes: prt:PART
Traits: Part[PART]
Attribute-Function: idle -> {prt};identify -> {prt};
Parameter-Specifications:
     Rec: prt;
     Sensed: prt;
Transition-Specifications:
     R1: <idle,identify>; Sensed[prt](true); true => true;
     R2: <identify,idle>; Rec[prt](true); true => true;
Time-Constraints:
     TCVar1: R1, Rec, [0, 5], {};
end
```

Figure 123: Vision System TROM class – class specification

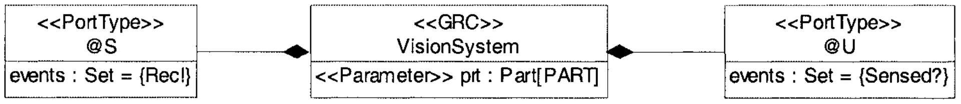


Figure 124: Vision System TROM class – class diagram

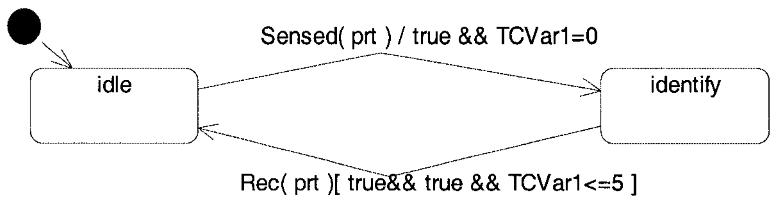

Sensed(prt) / true && TCVar1=0
idle
identify
Rec(prt)[true&& true && TCVar1<=5]

Figure 125: Vision System TROM class – state chart diagram

The Robot Class with Parameterized Events

The robot class waits until the vision system has recognized a part, and then saves the information carried by the parameter into a queue for later identification.

```
Class Robot [@D, @C, @E]
Events: Rec?@C, LeftPick!@D, RightPick!@D, Insert,
LeftPlace!@E, RightPlace!@E, LPopStack, FreeRight
States: *S1, S2, S3, S4, S5, S6
Attributes: rPrt:PART; lPrt:PART; inStack:PStack;
         inQueue:PQueue; prt:PART
Traits: Part[PART],Stack[PART,PStack],Queue[PART,PQueue]
Attribute-Function: S1 -> {inQueue, rPrt, prt};S2 -> {lPrt,
         inQueue, rPrt, prt};S3 -> {rPrt, inQueue, prt};S4 ->
         {lPrt, inQueue, prt};S5 -> {lPrt, inStack, inQueue,
         prt};S6 -> {inStack};
Parameter-Specifications:
     Rec: prt;
Transition-Specifications:
     R1: <S1,S1>; Rec[prt](true); true =>
         inQueue'=append(prt,inQueue);
     R2: <S1,S2>; LeftPick[](true); len(inQueue)>0 =>
         lPrt'=head(inQueue)&inQueue'=tail(inQueue);
     R3: <S2,S3>; RightPick[](true); len(inQueue)>0 =>
         rPrt'=head(inQueue)&inQueue'=tail(inQueue);
     R4: <S2,S2>; Rec[prt](true); true =>
         inQueue'=append(prt,inQueue);
     R5: <S3,S6>; Insert[](true); rPrt=lPrt =>
         inStack'=push(rPrt,inStack);
     R6: <S3,S4>; LeftPlace[](true); !(lPrt=rPrt) =>
         lPrt'=nullpart();
     R7: <S3,S3>; Rec[prt](true); true =>
         inQueue'=append(prt,inQueue);
     R8: <S4,S1>; RightPlace[](true); isEmpty(inStack) =>
         rPrt'=nullpart();
     R9: <S4,S5>; LPopStack[](true); !(isEmpty(inStack)) =>
         lPrt'=top(inStack)&inStack'=pop(inStack);
     R10: <S4,S4>; Rec[prt](true); true =>
         inQueue'=append(prt,inQueue);
     R11: <S5,S2>; RightPlace[](true); true =>
         rPrt'=nullpart();
     R12: <S5,S5>; Rec[prt](true); true =>
         inQueue'=append(prt,inQueue);
     R13: <S6,S2>; FreeRight[](true); true => rPrt'=nullpart();
Time-Constraints:
     TCVar1: R1, LeftPick, [0, 2], {};
     TCVar6: R8, LeftPick, [0, 1], {};
```

```
        TCVar2: R4, RightPick, [0, 2], {};
        TCVar7: R11, RightPick, [0, 1], {};
        TCVar5: R3, LeftPlace, [2, 3], {};
        TCVar4: R3, RightPlace, [3, 4], {};
        TCVar3: R3, RightPlace, [3, 4], {};
    end
```

Figure 126: Robot TROM class – class specification

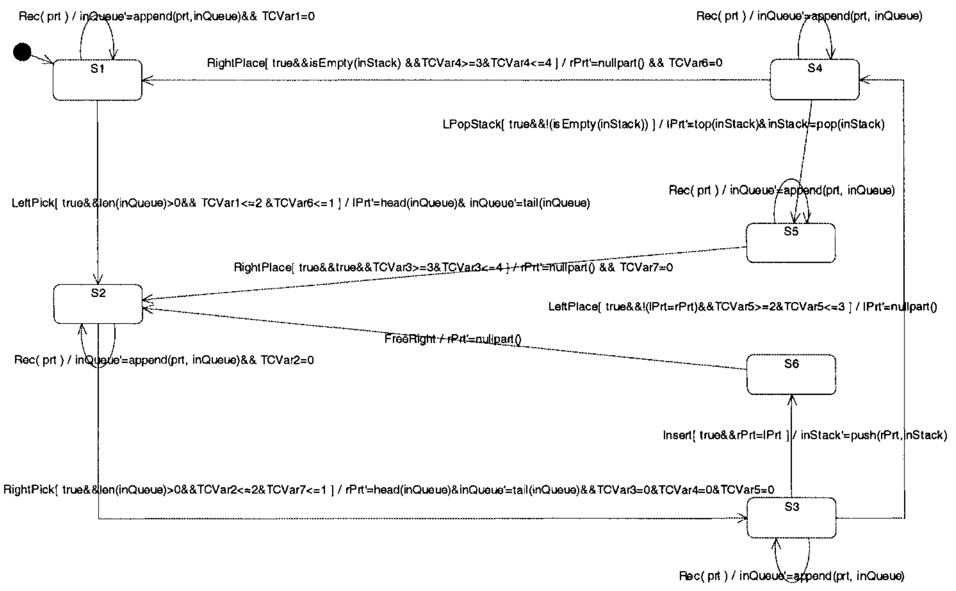

Rec(prt) / inQueue'=append(prt,inQueue)&& TCVar1=0
Rec(prt) / inQueue'=append(prt, inQueue)
S1
RightPlace[true&&isEmpty(inStack) &&TCVar4>=3&TCVar4<=4] / rPrt'=nullpart() && TCVar6=0
S4
LPopStack[true&&!(isEmpty(inStack))] / lPrt'=top(inStack)& inStack'=pop(inStack)
Rec(prt) / inQueue'=append(prt, inQueue)
LeftPick[true&&len(inQueue)>0&& TCVar1<=2 &TCVar6<=1] / lPrt'=head(inQueue)& inQueue'=tail(inQueue)
S5
RightPlace[true&&true&&TCVar3>=3&TCVar3<=4] / rPrt'=nullpart() && TCVar7=0
S2
LeftPlace[true&&!(lPrt=rPrt)&&TCVar5>=2&TCVar5<=3] / lPrt'=nullpart()
FreeRight / rPrt'=nullpart()
Rec(prt) / inQueue'=append(prt, inQueue)&& TCVar2=0
S6
Insert[true&&rPrt=lPrt] / inStack'=push(rPrt, inStack)
RightPick[true&&len(inQueue)>0&&TCVar2<=2&TCVar7<=1] / rPrt'=head(inQueue)&inQueue'=tail(inQueue)&&TCVar3=0&TCVar4=0&TCVar5=0
S3
Rec(prt) / inQueue'=append(prt, inQueue)

Figure 127: Robot TROM class – state chart diagram

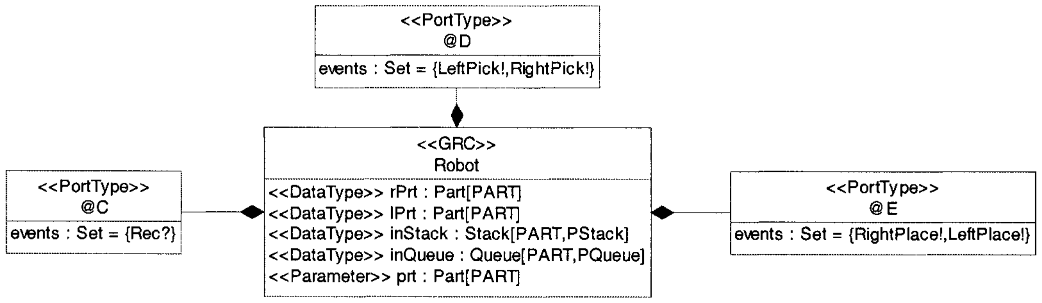

<<PortType>>
@D
events : Set = {LeftPick!,RightPick!}
<<GRC>>
Robot
<<DataType>> rPrt : Part[PART]
<<DataType>> lPrt : Part[PART]
<<DataType>> inStack : Stack[PART,PStack]
<<DataType>> inQueue : Queue[PART,PQueue]
<<Parameter>> prt : Part[PART]
<<PortType>>
@C
events : Set = {Rec?}
<<PortType>>
@E
events : Set = {RightPlace!,LeftPlace!}

Figure 128: Robot TROM class – class diagram

The Tray Class with Parameterized Events

The tray class does not define the parameter *prt* since the robot ensures that the two parts placed on the tray are of different types. The tray only assembles the parts.

```
Class Tray [@W]
Events: LeftPlace?@W, RightPlace?@W, Trash
```

```
States: *idle, wait, loading
Attributes:
Traits:
Attribute-Function: idle -> {};wait -> {};loading -> {};
Parameter-Specifications:

Transition-Specifications:
     R1: <idle,wait>; LeftPlace[](true); true => true;
     R2: <wait,loading>; RightPlace[](true); true => true;
     R3: <loading,idle>; Trash[](true); true => true;
Time-Constraints:
     TCVar1: R2, Trash, [1, 2], {};
end
```

Figure 129: Tray TROM class – class specification

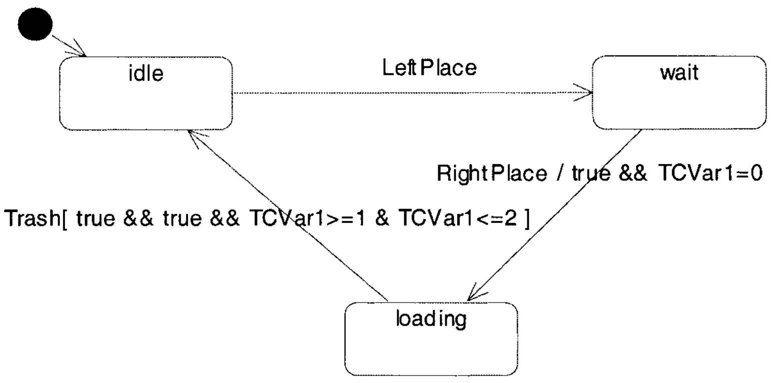

idle
LeftPlace
wait
RightPlace / true && TCVar1=0
Trash[true && true && TCVar1>=1 & TCVar1<=2]
loading

Figure 130: Tray TROM class – state chart diagram

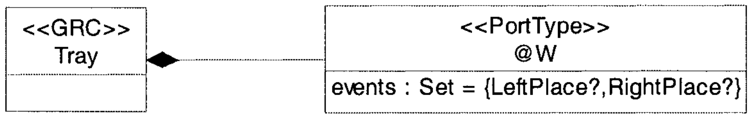


Figure 131: Tray TROM class – class diagram

The Sample Simulation Event List with Parameterized Events

The following is the sample simulation event list. It sends the same information as the above self-controlled user model to the system.

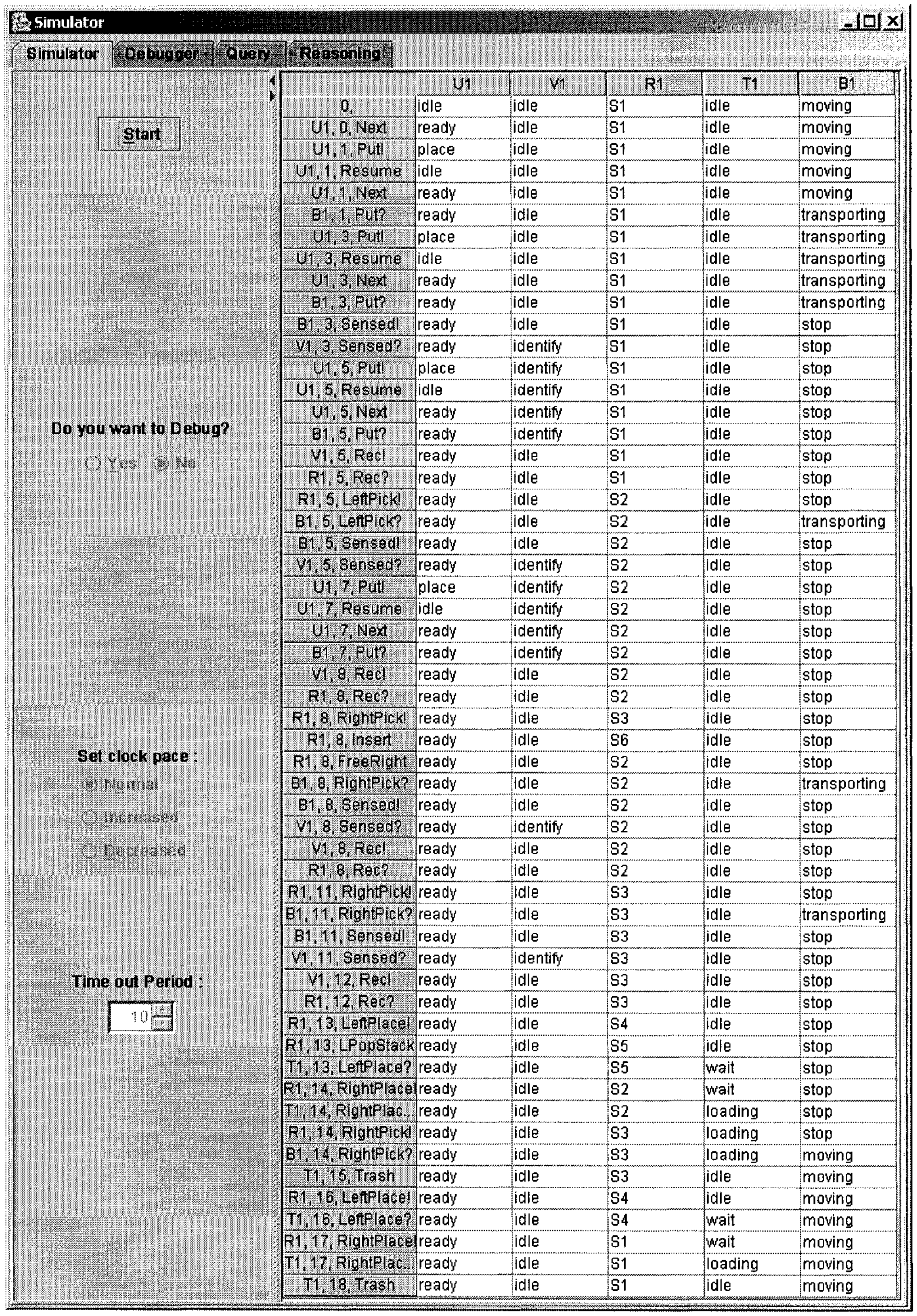
```
SEL: BasicRobot
      U1, Put[prt=dish()], @VS1, 3;
      U1, Put[prt=dish()], @VS1, 5;
      U1, Put[prt=cup()], @VS1, 7;
      U1, Put[prt=cup()], @VS1, 9;
end
```

Figure 132: Sample Simulation Event List

The Simulation Result with Parameterized Events

Figure 133 displays the simulation results of the buffered belt model. The successful simulation result and the successful simulation of the self-controlled model demonstrate that the revised robotics model with parameterized events is correct, and that the simulator can successfully model with Parameterized Events.

This chapter described two well-designed models for the robotics system. The models provided the base data for comparing different design solutions in terms of a system's functional complexity. The next chapter will describe the functionality measurement of the simulation results from two models without parameterized events.



Do you want to Debug?

Yes [✓ No]

Set clock pace :

[✓ Normal] Increased Decreased

Time out Period : 10

	U1	V1	R1	T1	B1
0,	idle	idle	S1	idle	moving
U1, 0, Next	ready	idle	S1	idle	moving
U1, 1, Put!	place	idle	S1	idle	moving
U1, 1, Resume	idle	idle	S1	idle	moving
U1, 1, Next	ready	idle	S1	idle	moving
B1, 1, Put?	ready	idle	S1	idle	transporting
U1, 3, Put!	place	idle	S1	idle	transporting
U1, 3, Resume	idle	idle	S1	idle	transporting
U1, 3, Next	ready	idle	S1	idle	transporting
B1, 3, Put?	ready	idle	S1	idle	transporting
B1, 3, Sensed!	ready	idle	S1	idle	stop
V1, 3, Sensed?	ready	identify	S1	idle	stop
U1, 5, Put!	place	identify	S1	idle	stop
U1, 5, Resume	idle	identify	S1	idle	stop
U1, 5, Next	ready	identify	S1	idle	stop
B1, 5, Put?	ready	identify	S1	idle	stop
V1, 5, Rec!	ready	idle	S1	idle	stop
R1, 5, Rec?	ready	idle	S1	idle	stop
R1, 5, LeftPick!	ready	idle	S2	idle	stop
B1, 5, LeftPick?	ready	idle	S2	idle	transporting
B1, 5, Sensed!	ready	idle	S2	idle	stop
V1, 5, Sensed?	ready	identify	S2	idle	stop
U1, 7, Put!	place	identify	S2	idle	stop
U1, 7, Resume	idle	identify	S2	idle	stop
U1, 7, Next	ready	identify	S2	idle	stop
B1, 7, Put?	ready	identify	S2	idle	stop
V1, 8, Rec!	ready	idle	S2	idle	stop
R1, 8, Rec?	ready	idle	S2	idle	stop
R1, 8, RightPick!	ready	idle	S3	idle	stop
R1, 8, Insert	ready	idle	S6	idle	stop
R1, 8, FreeRight	ready	idle	S2	idle	stop
B1, 8, RightPick?	ready	idle	S2	idle	transporting
B1, 8, Sensed!	ready	idle	S2	idle	stop
V1, 8, Sensed?	ready	identify	S2	idle	stop
V1, 8, Rec!	ready	idle	S2	idle	stop
R1, 8, Rec?	ready	idle	S2	idle	stop
R1, 11, RightPick!	ready	idle	S3	idle	stop
B1, 11, RightPick?	ready	idle	S3	idle	transporting
B1, 11, Sensed!	ready	idle	S3	idle	stop
V1, 11, Sensed?	ready	identify	S3	idle	stop
V1, 12, Rec!	ready	idle	S3	idle	stop
R1, 12, Rec?	ready	idle	S3	idle	stop
R1, 13, LeftPlace!	ready	idle	S4	idle	stop
R1, 13, LPopStack	ready	idle	S5	idle	stop
T1, 13, LeftPlace?	ready	idle	S5	wait	stop
R1, 14, RightPlace!	ready	idle	S2	wait	stop
T1, 14, RightPlac...	ready	idle	S2	loading	stop
R1, 14, RightPick!	ready	idle	S3	loading	stop
B1, 14, RightPick?	ready	idle	S3	loading	moving
T1, 15, Trash	ready	idle	S3	idle	moving
R1, 16, LeftPlace!	ready	idle	S4	idle	moving
T1, 16, LeftPlace?	ready	idle	S4	wait	moving
R1, 17, RightPlace!	ready	idle	S1	wait	moving
T1, 17, RightPlac...	ready	idle	S1	loading	moving
T1, 18, Trash	ready	idle	S1	idle	moving

Figure 133: The Simulation Result of Buffered Belt modeling

Chapter 8

Simulation-Based Measurement of System Functionality

This chapter measures the functional complexity of real-time reactive systems based on design animation. Software measurement quantifies the attributes of software in order to characterize them by clearly defined rules. The proposed measurement of functionality quantifies the design's attribute functional complexity in terms of the amount of information interchanged during the system's simulation. Our goal is to compare the functional complexity of different design solutions based on the measurement data.

8.1 Functionality Measure

The functionality measure *FC* is defined as the information content of the interactions in one scenario. Information theory based software measurement is used to quantify objectively the software functionality in terms of the amount of information interchanged between software objects. The information content of the interactions is synonymous with the complexity of the product's functionality, so measuring the amount of information assesses the performance of the software system.

Let S be a subsystem consisting of $O_1, \dots O_n$ reactive objects. Let *Comp*(S) be a set of all computations in one period of time (i.e., the time between two consecutive initial states). Let *SQ'*(S) be the projection of *Comp*(S) on the signals ($\sigma_1, \dots \sigma_n$).

For functional complexity measurement purposes, we need the projection of *SQ'*(S) on the sequence of events *Events*(S) = $\{e_1, \dots e_n\}$. These events present the functions needed to perform the work in the specified period of time, and the sequence preserves their order in time.

The functional complexity in a time period is defined as an average amount of information in the corresponding sequence *Events*(S). We apply the concepts of information theory to measure the amount of work performed in a time slice by the system in terms of the amount of information in the *Events*(S) sequence. We use the *excess-entropy* information theory's measure C to quantify the amount of information.

8.2 Formulas

We based our version of the excess-entropy measure on the empirical distribution of events within a sequence. The probability p_i of the i^{th} occurring event is equal to the percentage of total event occurrences. The probability p_i is calculated as $p_i = \frac{f_i}{NE}$ where f_i is the number of occurrences of the i^{th} event and NE is the total number of events in the sequence. The classical excess-entropy calculation is given by the formula $C = \sum H_j - H$, where $\mathrm{H} = \sum p_i \log_2 p_i$ is the entropy for the system, and H_j is the entropy calculated locally for each one of the participating objects.

Let E be the number of different events that happen in the system in a specific time interval T, E_{oj} be the number of those events related to the object O_j (in T), and f_i^{Oj} be the frequency of the event $i \in E_{Oj}$. We set T to be the response time for a scenario.

We define our measure of the functional complexity FC as a difference between the sum of the objects' entropy and the system's entropy:

$$FC = \sum_{j=1}^{n} H_{Oj} - H$$

Where

$$H_{Oj} = -\sum_{i=1}^{E_{Oj}} \frac{f_i^{Oj}}{NE_{Oj}} \log_2 \frac{f_i^{Oj}}{NE_{Oj}}$$

$$H = \sum_{i=1}^{E} \frac{f_i}{NE} \log_2 \frac{f_i}{NE}$$

The FC measure is intended to be used on an ordinal scale. The first step in an ordinal measurement's procedure is to determine an ordering relation for the objects. The second is to assign a number sequence that preserves the order of the objects; i.e., the FC measure is intended to order the performance of real-time reactive systems in a time period in relation to their functional complexity. From our empirical understanding, the performance of a system S_1 whose functional complexity indicates higher average information content in a time interval t than the system S_2 should, on the whole, be more complex than another with a lower average information content. Formally, the ordinal scale property can be expressed as follows:

$$S_1 \succcurlyeq S_2 \Leftrightarrow FC(S_1) \geq FC(S_2)$$

The functionality measurement has been empirically validated on the Robotics case study.

8.3 Illustration on the Robotics Case Study

In this section, the results from the empirical validation of the functionality measurement are reported and analyzed. The empirical validation is a process of establishing the accuracy of the software measurement by empirical means. In case of the comparison measurement, the empirical validation identifies the extent to which a measure characterizes a stated attribute by a simple test against reality.

```
SEL: BasicRobot
        PutD, U1, @VS1, 1;
        PutC, U1, @VS1, 8;
end
```

Figure 134: Simulation Event List 1

```
SEL: BasicRobot
        PutD, U1, @VS1, 1;
        PutD, U1, @VS1, 8;
        PutC, U1, @VS1, 15;
        PutC, U1, @VS1, 22;
end
```

Figure 135: Simulation Event List 2

```
SEL: BasicRobot
        PutD, U1, @VS1, 1;
        PutD, U1, @VS1, 8;
        PutD, U1, @VS1, 15;
        PutC, U1, @VS1, 22;
        PutC, U1, @VS1, 29;
        PutC, U1, @VS1, 36;
end
```

Figure 136: Simulation Event List 3

We have calculated the functionality measure *FC* for each one of the Robotics models, and for each one of the system events lists *SEL*1, *SEL*2, *SEL*3 designed to gradually increase in functionality. We have also considered the structural complexity of the designs, as described in [Orm02]. The static architectural complexity (*AC*) indicates the level of external coupling within a given architecture. Measuring *AC* would give an objective base for comparing different design models in terms of their *FC* values. The following table reports the *FC* and *AC* values:

	SEL1	**SEL2**	**SEL3**	**AC**
Original	7.96	8.01	8.16	0.35
Selft-controlled User(Non-parameterized Events)	8.07	8.11	8.34	0.38
Buffered Belt(Non-parameterized Events)	7.96	8.01	8.14	0.35
Selft-controlled User(parameterized Events)	6.86	7.26	7.20	0.38
Buffered Belt(parameterized Events)	5.91	6.19	6.13	0.35

Table 3: FC and AC values of three Robotics models

For a given design, the increasing complexity of *SEL* files corresponds to the increasing values of *FC*. From the other side, for a given *SEL* file, the higher behavioral and structural complexity of the *GRC*s in the self-controlled user model results in higher *FC* values. Based on the reported results, the design Buffered Belt model delivered the best performance results because its structural and behavioral complexities are lower than in the self-controlled user model. The reported results were expected from our empirical understanding, and therefore prove experimentally the validity of the measures.

Chapter 9

Conclusions and Future Work

The present thesis addresses the problem of automatic validation of real-time reactive systems at the design phase before the implementation. We assume that such systems are modeled as timed labeled transition systems, following the TROM methodology with parameterized events. In the previous work on the simulation, there existed some problems in the simulator algorithm, which result in Train-Gate-Controller and Robotics system case study to be simulated unsuccessfully. The goal of this thesis is to improve flaws related to time conflicts in previous models, separate the data model module from the validation tool, and add support to parameterized events into the simulator. As system gets larger, the performance assessment for mission critical applications is more important than ever. This thesis proposes a performance assessment of systems in terms to their functionality, based on simulation. We have illustrated our approach on the Robotics case study.

9.1 Work Synthesis

In this thesis, we discussed the time conflict existing in the previous simulator implementation and improved the simulation algorithm. We analyzed all possible situations to ensure that our algorithm was correct. After we fixed the algorithm, the validation tool was ready for working. We verified the correctness of the validation tool through the Train-Gate-Controller case study.

Next, to provide more extensibility and flexibility to the validation tool, we re-organized the structure by applying the Façade Pattern for separating the data model from the interpreter and Simulator modules.

After we fixed the internal flaws and adjusted the global structure, we start to extend the Validation tool's ability to support simulation for real-time reactive system modeling based on TROM formalism [Ach95] with parameterized events. We reengineered the interpreter, upgraded the simulator, and verified the new validation tool by simulating the TGC model with Parameterized Events.

So far, we finished upgrading the Validation tool. We took advantage of it to remodel the Robotics system. Through the simulation result, we found out the problems in the original modeling, gave two solutions, and verified our solutions by the Validation tool.

Finally, we proposed a new functionality measurement in real-time reactive systems based on design simulation [ALOS03]. Through gathering measurement data based on their simulation results, we can reach the goal of early functional complexity predication, and compare design solutions in terms of their functionality.

9.2 Future Work

9.2.1 Parameterized Events

After the Parameterized Events were introduced into the Interpreter and Simulator, the Debugger, Query, and Reasoning System need to be upgraded in the future.

9.2.2 Simulator

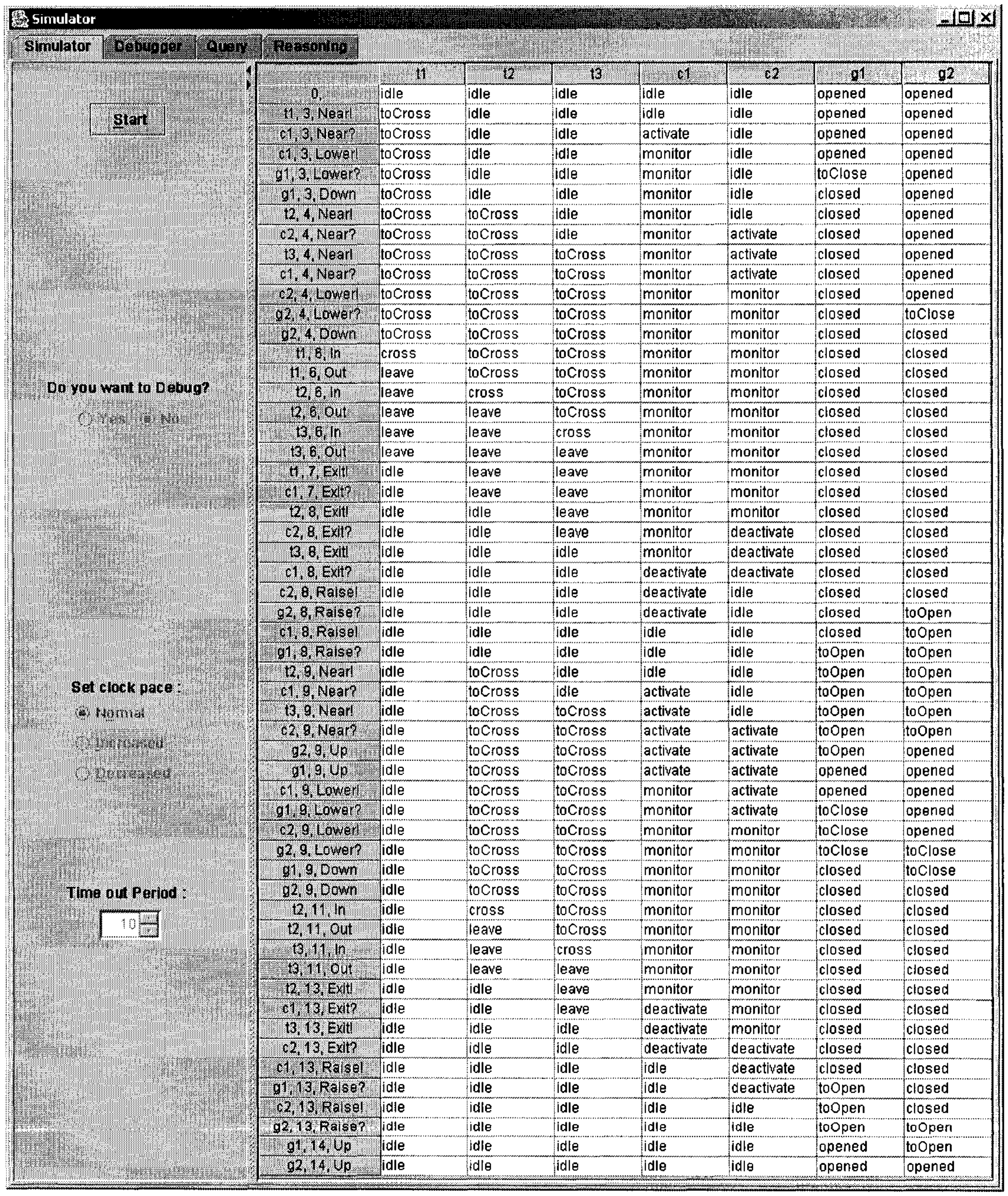
After this thesis work, the Simulator is upgraded to allow simulating TGC and Robotics systems. However, there is a known limitation to the new Simulator when it simulates the TGC system. When a train comes out a gate, it is unable for the train to start a new session immediately, i.e., if a train needs to pass through two gates, the train must have exited the first gate before it approachs the second gate. Formally, the train cannot exit the first gate and send *Near* messasge to the second gate at same time.

We will use the specification of TGC system showed in the Chapter 6 to explain the case. Figure 137 is the SEL to expose the problem.

```
SEL: TCG
   t1, Near, @C1, 3;
   t2, Near, @C2, 4;
   t3, Near, @C1, 4;
   t1, Near, @C2, 7;
   t2, Near, @C1, 9;
   t3, Near, @C2, 9;
end
```

Figure 137: Simulation Event List

Figure 138 shows the Simulation Result. The train t1 approachs gate 1 at time 3 by sending *Near* to controller 1. It exits gate1 at time 7 (the *Exit* event is marked by the balck arrow). From the SEL, we know train t1 will send *Near* message at time 7, but it is not handled by the simulator.



Do you want to Debug?
Yes [✓ No]

Set clock pace :
[✓ Normal]
Increased
Decreased

Time out Period : 10

	t1	t2	t3	c1	c2	g1	g2
0,	idle	idle	idle	idle	idle	opened	opened
t1, 3, Near!	toCross	idle	idle	idle	idle	opened	opened
c1, 3, Near?	toCross	idle	idle	activate	idle	opened	opened
c1, 3, Lower!	toCross	idle	idle	monitor	idle	opened	opened
g1, 3, Lower?	toCross	idle	idle	monitor	idle	toClose	opened
g1, 3, Down	toCross	idle	idle	monitor	idle	closed	opened
t2, 4, Near!	toCross	toCross	idle	monitor	idle	closed	opened
c2, 4, Near?	toCross	toCross	idle	monitor	activate	closed	opened
t3, 4, Near!	toCross	toCross	toCross	monitor	activate	closed	opened
c1, 4, Near?	toCross	toCross	toCross	monitor	activate	closed	opened
c2, 4, Lower!	toCross	toCross	toCross	monitor	monitor	closed	opened
g2, 4, Lower?	toCross	toCross	toCross	monitor	monitor	closed	toClose
g2, 4, Down	toCross	toCross	toCross	monitor	monitor	closed	closed
t1, 6, In	cross	toCross	toCross	monitor	monitor	closed	closed
t1, 6, Out	leave	toCross	toCross	monitor	monitor	closed	closed
t2, 6, In	leave	cross	toCross	monitor	monitor	closed	closed
t2, 6, Out	leave	leave	toCross	monitor	monitor	closed	closed
t3, 6, In	leave	leave	cross	monitor	monitor	closed	closed
t3, 6, Out	leave	leave	leave	monitor	monitor	closed	closed
t1, 7, Exit!	idle	leave	leave	monitor	monitor	closed	closed
c1, 7, Exit?	idle	leave	leave	monitor	monitor	closed	closed
t2, 8, Exit!	idle	idle	leave	monitor	monitor	closed	closed
c2, 8, Exit?	idle	idle	leave	monitor	deactivate	closed	closed
t3, 8, Exit!	idle	idle	idle	monitor	deactivate	closed	closed
c1, 8, Exit?	idle	idle	idle	deactivate	deactivate	closed	closed
c2, 8, Raise!	idle	idle	idle	deactivate	idle	closed	closed
g2, 8, Raise?	idle	idle	idle	deactivate	idle	closed	toOpen
c1, 8, Raise!	idle	idle	idle	idle	idle	closed	toOpen
g1, 8, Raise?	idle	idle	idle	idle	idle	toOpen	toOpen
t2, 9, Near!	idle	toCross	idle	idle	idle	toOpen	toOpen
c1, 9, Near?	idle	toCross	idle	activate	idle	toOpen	toOpen
t3, 9, Near!	idle	toCross	toCross	activate	idle	toOpen	toOpen
c2, 9, Near?	idle	toCross	toCross	activate	activate	toOpen	toOpen
g2, 9, Up	idle	toCross	toCross	activate	activate	toOpen	opened
g1, 9, Up	idle	toCross	toCross	activate	activate	opened	opened
c1, 9, Lower!	idle	toCross	toCross	monitor	activate	opened	opened
g1, 9, Lower?	idle	toCross	toCross	monitor	activate	toClose	opened
c2, 9, Lower!	idle	toCross	toCross	monitor	monitor	toClose	opened
g2, 9, Lower?	idle	toCross	toCross	monitor	monitor	toClose	toClose
g1, 9, Down	idle	toCross	toCross	monitor	monitor	closed	toClose
g2, 9, Down	idle	toCross	toCross	monitor	monitor	closed	closed
t2, 11, In	idle	cross	toCross	monitor	monitor	closed	closed
t2, 11, Out	idle	leave	toCross	monitor	monitor	closed	closed
t3, 11, In	idle	leave	cross	monitor	monitor	closed	closed
t3, 11, Out	idle	leave	leave	monitor	monitor	closed	closed
t2, 13, Exit!	idle	idle	leave	monitor	monitor	closed	closed
c1, 13, Exit?	idle	idle	leave	deactivate	monitor	closed	closed
t3, 13, Exit!	idle	idle	idle	deactivate	monitor	closed	closed
c2, 13, Exit?	idle	idle	idle	deactivate	deactivate	closed	closed
c1, 13, Raise!	idle	idle	idle	idle	deactivate	closed	closed
g1, 13, Raise?	idle	idle	idle	idle	deactivate	toOpen	closed
c2, 13, Raise!	idle	idle	idle	idle	idle	toOpen	closed
g2, 13, Raise?	idle	idle	idle	idle	idle	toOpen	toOpen
g1, 14, Up	idle	idle	idle	idle	idle	opened	toOpen
g2, 14, Up	idle	idle	idle	idle	idle	opened	opened

Figure 138: Simulation Result

Creating the internal simulation event list in the simulator posed a problem. When the simulator began, it created an initial internal simulation event list with an environmental event list, as described by the SEL. Later, the simulator created new events, which were inserted into the internal simulation event list according to the specification. If the event is internal, it will be inserted into the internal simulation event list as the first issuing event at the specified time. If the event is an input or an output event, it will be appended to the tail of those events that will be issued at the specified time. For this case, event *E1 – t1, near, c2, 7* will be scheduled prior to the event *E2 – t1, exit, c1, 7* in the internal simulation event list. However, before *E2* is issued, the state of t1 changes to *leave*. Therefore, according to the specification, *E1* will not be issued. This problem can be solved by modifying the insertion method of events and will be left to the future work.

Bibliography

[AAM98] V. S. Alagar, R. Achuthan, and D. Muthianyen. *TROMLAB: A software development environment for real-time reactive system. Submitted for publication in ACM Transactions on Software Engineering and Methodology* (First version 1996, revised 2001), submitted for publication

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

[All84] J. Allen. Towards a general theory of action and time. *Artificial intelligence(23)*, 1984.

[ALOS03] V. S. Alagar, S. H. Liu, O. Ormandjieva, J. Shen. *Performance Assessment in Real-Time Reactive Systems.* In the proceedings of the 7th IASTED International Conference on Software Engineering and Applications (SEA 2003)

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

[GOF94] E. Gamma, R. Helm, R. Jhonson, J. Vlissides. *Design Patterns.* August, 1994

[Hai99] G. Haidar. *Reasoning System for Real-Time Reactive Systems.* Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, December 1999

[Hay01] M. Haydar. *Parameterized Events for Designing Real-Time Reactive Systems.* Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, February 2001

[Mut96] D.Muthiayen. *Animation and Formal Verification of Real-Time Reactive Systems in an Object-Oriented Environment.* Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, October 1996

[Orm02] O. Ormandjieva, *Deriving New Measurements for Real-Time Reactive Systems*. PhD. Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 2002

[Sri99] V. Srinivasan. *Graphical User Interface for TROMLIB Environment.* Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, December 1999.

[Tao96] H. Tao. *Static Analyzer: A Design Tool for TROM.* Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, August 1996

Appendix A

Code Optimization

The project originally had three directories: Gui, Interpreter, and IntSim, and the corresponding source files were in these directories. Although files are in different directories, they were not organized into different packages, i.e., they were in the same global namespace. The previous workflow was: first, compiled the source code in their directories; second, copied the class file to the Gui directory; third, run the program in the Gui directory. It made the project difficult to manage and there was name conflict in the project files, too. For example, there were Portlist.class and portlist.class in Gui directory. Because the original development environment was Unix, there was no problem. But it caused problems in the Windows operation system because file name in Windows was not case sensitive. That is, it is operation system-dependent. Therefore, it is necessary to optimize the code to get a real platform-independent project. Organizing them into different Java packages to make them exist in their own name space is a good choice. The following figure is the new structure of the Validation tool.

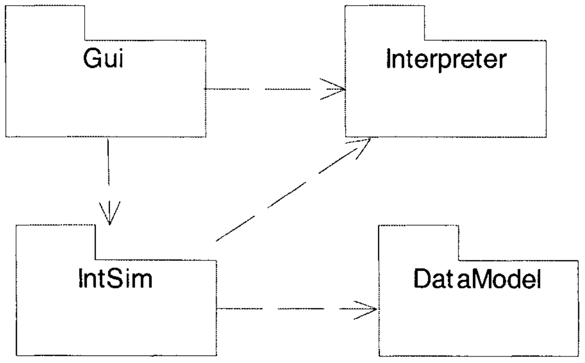


Figure 139: The logical structure of the Validation tool

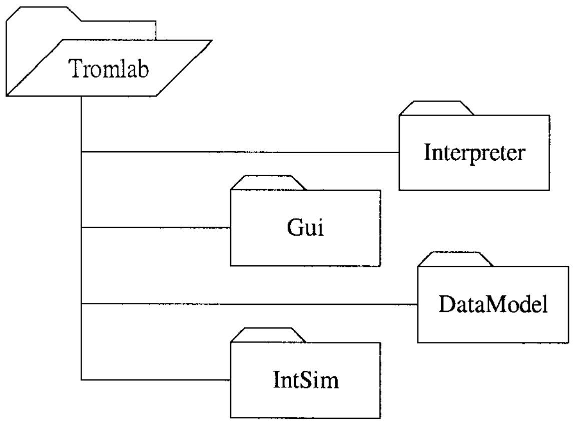
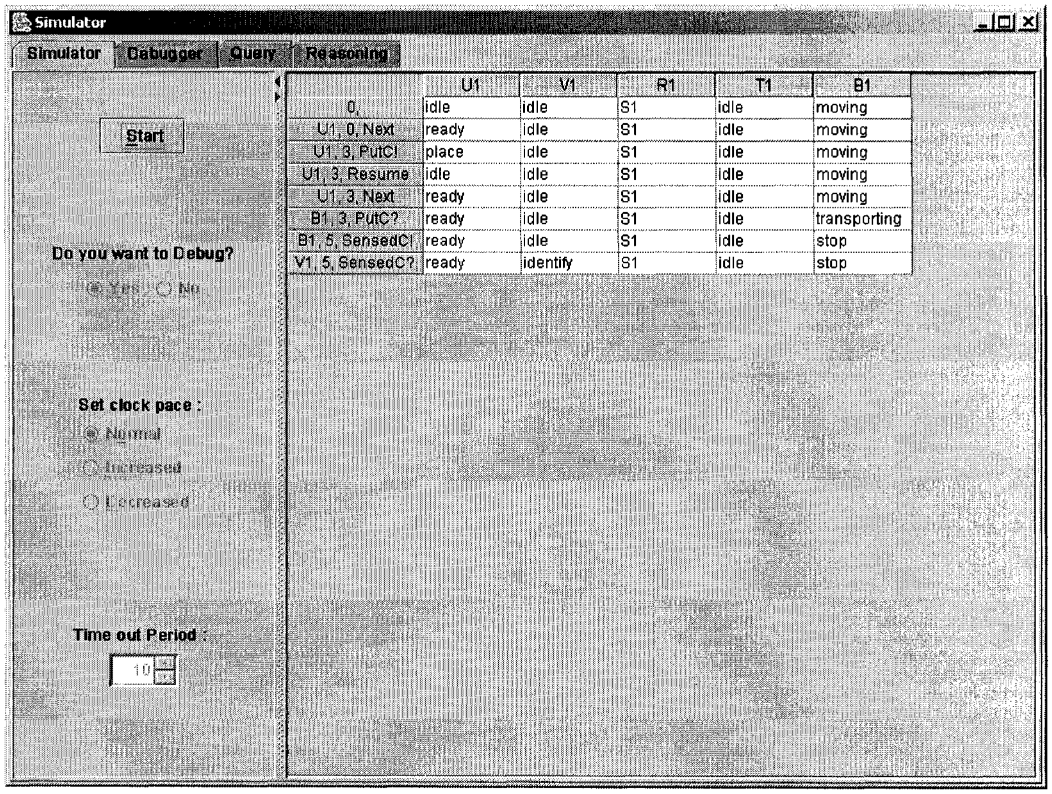


Figure 140: The physical structure of the Validation tool

Appendix B

New Graphical User Interface

Because the output areas of the Debugger, Reasoning System are too small, we re-design the graphical user interface of the Simulator. The new GUI larger output area, and user-friendly interface. The following are the new user interfaces of the Simulator.



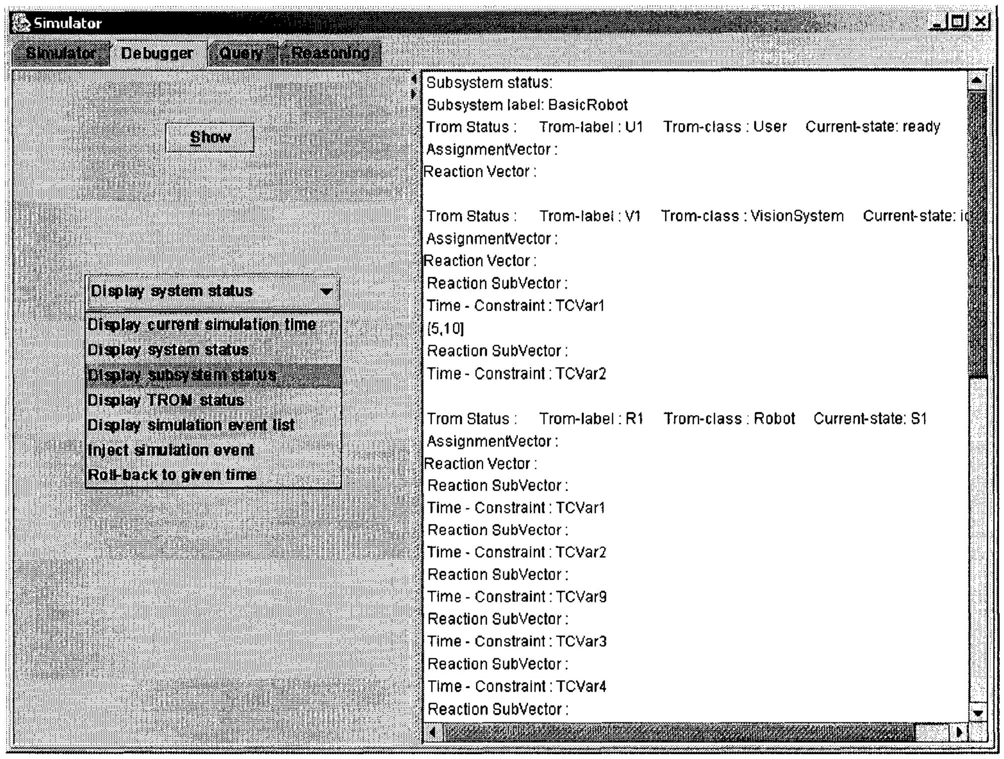
	U1	V1	R1	T1	B1
0,	idle	idle	S1	idle	moving
U1, 0, Next	ready	idle	S1	idle	moving
U1, 3, PutC!	place	idle	S1	idle	moving
U1, 3, Resume	idle	idle	S1	idle	moving
U1, 3, Next	ready	idle	S1	idle	moving
B1, 3, PutC?	ready	idle	S1	idle	transporting
B1, 5, SensedC!	ready	idle	S1	idle	stop
V1, 5, SensedC?	ready	identify	S1	idle	stop

Do you want to Debug?
[✓ Yes] No

Set clock pace :
[✓ Normal]
Increased
Decreased

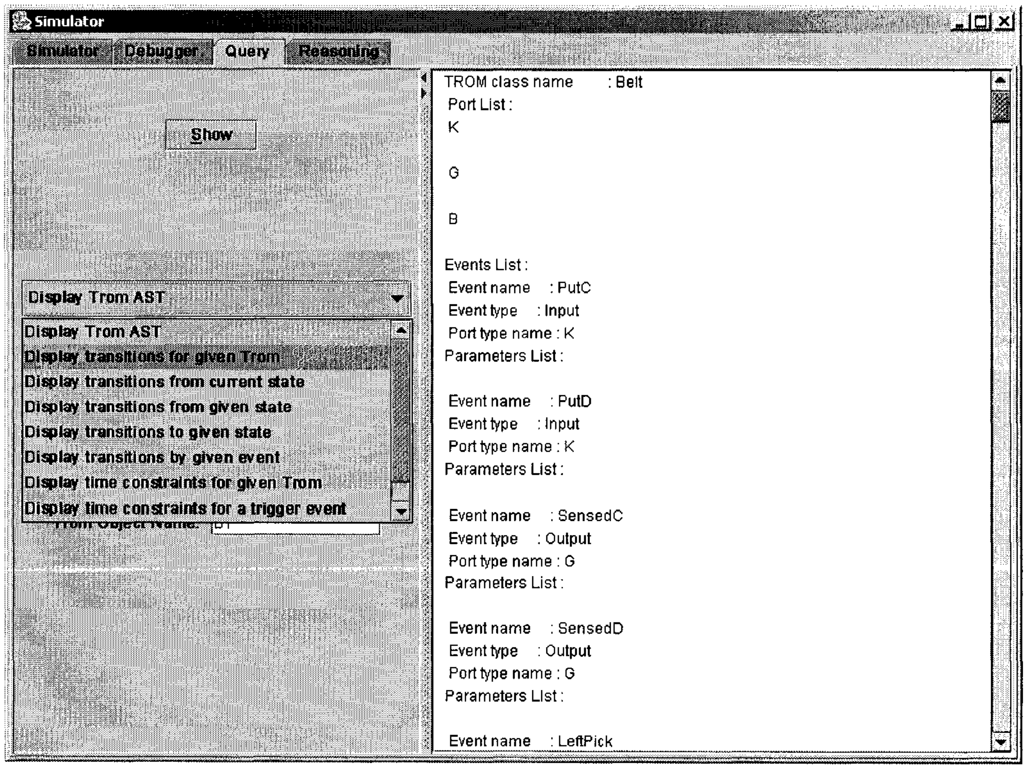
Time out Period : 10

Figure 141: The interface of the Simulator



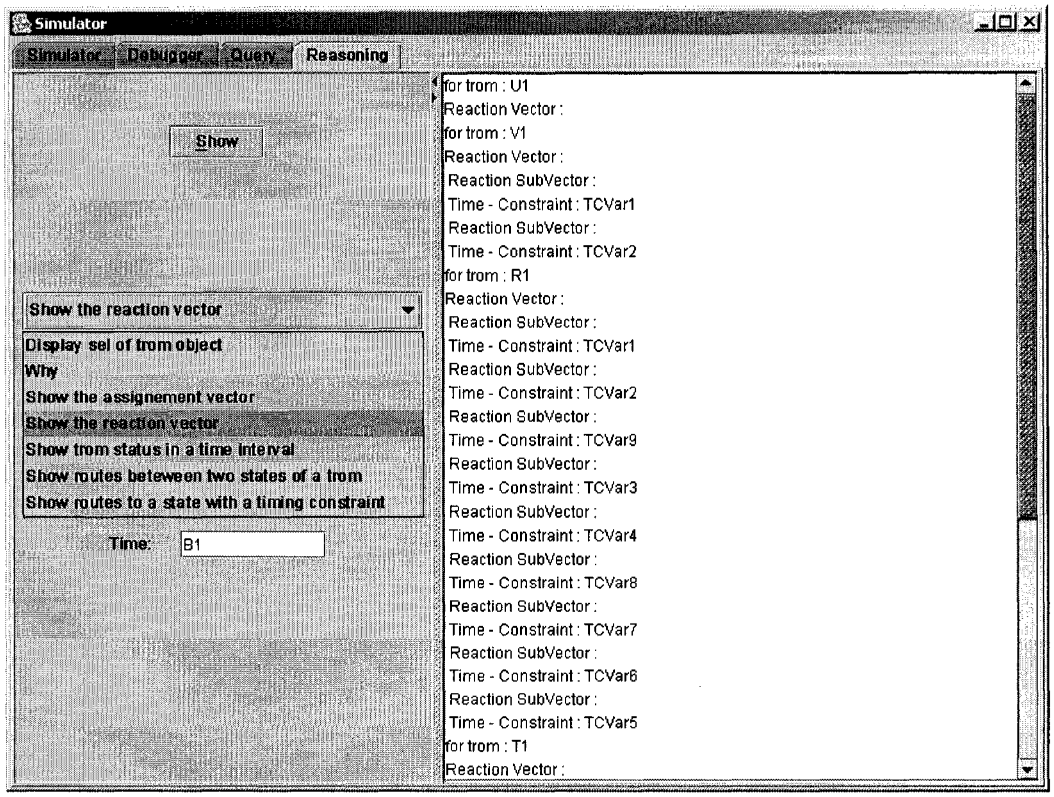
```
Subsystem status:
Subsystem label: BasicRobot
Trom Status :    Trom-label : U1    Trom-class : User    Current-state: ready
AssignmentVector :
Reaction Vector :

Trom Status :    Trom-label : V1    Trom-class : VisionSystem    Current-state: i
AssignmentVector :
Reaction Vector :
Reaction SubVector :
Time - Constraint : TCVar1
[5,10]
Reaction SubVector :
Time - Constraint : TCVar2

Trom Status :    Trom-label : R1    Trom-class : Robot    Current-state: S1
AssignmentVector :
Reaction Vector :
Reaction SubVector :
Time - Constraint : TCVar1
Reaction SubVector :
Time - Constraint : TCVar2
Reaction SubVector :
Time - Constraint : TCVar9
Reaction SubVector :
Time - Constraint : TCVar3
Reaction SubVector :
Time - Constraint : TCVar4
Reaction SubVector :
```

Figure 142: The interface of the Debugger


Simulator
Simulator
Debugger
Query
Reasoning
Show
Display Trom AST
Display Trom AST
Display transitions for given Trom
Display transitions from current state
Display transitions from given state
Display transitions to given state
Display transitions by given event
Display time constraints for given Trom
Display time constraints for a trigger event

```
TROM class name      : Belt
 Port List :
 K

 G

 B

Events List :
 Event name    : PutC
 Event type    : Input
 Port type name : K
Parameters List :

 Event name    : PutD
 Event type    : Input
 Port type name : K
Parameters List :

 Event name    : SensedC
 Event type    : Output
 Port type name : G
Parameters List :

 Event name    : SensedD
 Event type    : Output
 Port type name : G
Parameters List :

 Event name    : LeftPick
```

Figure 143: The interface of the Query


Simulator
Simulator
Debugger
Query
Reasoning
Show
Show the reaction vector
Display set of trom object
Why
Show the assignement vector
Show the reaction vector
Show trom status in a time interval
Show routes beteween two states of a trom
Show routes to a state with a timing constraint
Time:

B1

```
for trom : U1
Reaction Vector :
for trom : V1
Reaction Vector :
 Reaction SubVector :
 Time - Constraint : TCVar1
 Reaction SubVector :
 Time - Constraint : TCVar2
for trom : R1
Reaction Vector :
 Reaction SubVector :
 Time - Constraint : TCVar1
 Reaction SubVector :
 Time - Constraint : TCVar2
 Reaction SubVector :
 Time - Constraint : TCVar9
 Reaction SubVector :
 Time - Constraint : TCVar3
 Reaction SubVector :
 Time - Constraint : TCVar4
 Reaction SubVector :
 Time - Constraint : TCVar8
 Reaction SubVector :
 Time - Constraint : TCVar7
 Reaction SubVector :
 Time - Constraint : TCVar6
 Reaction SubVector :
 Time - Constraint : TCVar5
for trom : T1
Reaction Vector :
```

Figure 144: The interface of the Reasoning System

Appendix C

Software Document

Because there was no software document for this system in the past, it is time-consumed for newcomers to grasp the structure of the system. This software document tries to record some main classes in the Validation tool to assist subsequent researchers to understand the system. It still needs future researchers to continue this work.

Class AssignmentVector: To model the value of **attributes** of the Trom class defined in .trom file.

Attributes	
AssignmentList	assignment_list
attributelist	attribute_asts
LSLLibraryManager	llm
Methods	
	AssignmentVector(TROMclass statics, LSLLibraryManager lsl_lmgr) Constructor. Construct the assignmentlist with attributelsit. It traverse the attribute list of *statics,* and for each attribute, construct a new assignment that may be a Port type or a kind of Trait type. If it's a kind of trait type, *lsl_lmgr* will be responsible for creating the trait object of corresponding type.
	AssignmentVector(AssignmentVector av) Copy constructor.
Assignment	**get_assignment(String assignment_name)** Traverse the assignment list to find and return the assignment naming *assignment_name*
void	**set_assignment(Assignment assign)** Update the assignment in the assignment list

Class EventHandler: The most important class in the Simulator. It is responsible for handling outstanding events in the SimulationEventList

Attributes	
Subsystem	ss

SimulationEventList	sel
TimeManager	tm
EventScheduler	es
ReactionWindowManager	rwm
IDataSupport	ds
Methods	
	EventHandler() Default constructor
	EventHandler(Subsystem subsys, SimulationEventList selist, TimeManager tmgr, EventScheduler esched, ReactionWindowManager rwmgr, IDataSupport ids) Constructor
void	**handle_event(SimulationEvent se)** Handle outstanding event in the SimulationEventList *sel*, including saving history, passing parameters, evaluating port-, pre- and post-condition, handling reaction window, and scheduling new events.
trans_spec	**get_transition_spec(SimulationEvent se)** Get transition specification corresponding to *se*.

Class EventScheduler: The scheduler of simulating events. It controls the time order of when and what events should be fired. It is one of core classes of Simulator module.

Attributes	
Subsystem	ss
SimulationEventList	sel
TimeManager	tm
Tromclasslist	Trom_asts
SCSSimEv	Se_asts
Methods	
	EventScheduler() Default constructor
	EventScheduler(Subsystem subsys, SimulationEventList selist, TimeManager tmgr, Tromclasslist trom_ast_list, SCSSimEv simev_ast_list) Constructor
void	**schedule_batchmode_events()** Only "Output" and not constrained events can be scheduled in batch mode; others will be ignored. It traverses the SimEvList to extract events, which is type of "Output" and not constrained, to create SimulationEvent objects and insert SimulationEventList. Finally, invoke schedule_rendez_vous(…) to add events relative to these events to SimulationEventList.
void	**schedule_rendez_vous(SimulationEvent se)** It searchs all events associated with *se* and insert them into SimulationEventList. Get TromPortTuple object associated with *se*, and then obtain the corresponding TromPortTuple objects from link sdefined by PortLink objects. Create new SimulationEvent objects with these TromPortTuple objecs and same event, and then insert these SimulationEvent objects into SimulationEventList object.
void	**schedule_unconstrained_internal_events_from_initial_state()** It transfers the invocation to the internal function call: schedule_unconstrained_internal_events_from_initial_state(Subsystem subsys)
void	**schedule_unconstrained_internal_events_from_initial_state(Subsystem subsys)** Traverse Subsystem list. For each Subsytem object, traverse all Trom objects and invoke the mothed *schedule_unconstrained_internal_events_from_initial_state(Trom trom)* for each Trom object.

void	**schedule_unconstrained_internal_events_from_initial_state(Trom trom)** At the beginning of the system running, get all unconstrained internal events, whose states are same as the current state of *trom*, from TROMclass object; afterwards, create SimulationEvent objects with them and insert into SimulationEventList.
void	**schedule_unconstrained_internal_event(Trom trom)** Get all unconstrained internal events such as "Out" event, which fire the state same as the current state of *trom*, from TROMclass object; afterwards, create SimulationEvent objects with them and insert into SimulationEventList.

Class Int_Sim: Get AST and initialize all relative classes, then start the simulator

Attributes	
Subsystem	ss;
SimulationEventList	sel
TimeManager	tm
Simulator	sim
LSLLibraryManager	llm
LSLLibrarySupport	lls
ObjectModelSupport	oms
SubsystemModelSupport	sms
ConsistencyChecker	cc
QueryHandler	qh
TraceAnalyzer	ta
Debugger	dbg
AxiomGenerator	ag
Reasoning_system	rs
Methods	
	Int_Sim() To initialize all data members to null.
void	**Start(AST ast, SimulatorPane sPane, String d, String c, int t)** To create various objects by using AST object and construct a new object of class Simulator with these objects, and then execute the run() of this Simulator object to start the simulation.

Class LSLLibraryManager: To create instances of Traits defined in LSL library.

Attributes	
Methods	
Trait	**Trait new_lsl_trait(LSLtrait trait_ast, attribute attribute_ast)** Object creation method. It creates a kind of trait object according to the name described in *trait_ast.*
Trait	**copy_lsl_trait(Trait lsl_trait)** Trait object copy constructor. It creates a new trait object from an existing trait object.

Class LSLLibrarySupport: To implement function call defined in traits' definition.

Attributes	
Methods	
object	**evaluate_function_call(SimpleNode expr, Port pid, AssignmentVector asgn_vect)** To assert which trait object should receive this function call and then invoking corresponding following functions.
object	**evaluate_function_call_set(Trait tr, SimpleNode expr, Port pid, AssignmentVector asgn_vect)** Make a function call to Set object and return the result.
object	**evaluate_function_call_queue(Trait tr, SimpleNode expr, Part p, AssignmentVector asgn_vect)** Make a function call to Queue object and return the result.
object	**Object evaluate_function_call_stack(Trait tr, SimpleNode expr, Part p, AssignmentVector asgn_vect)** Make a function call to Stack object and return the result.
object	**Object evaluate_function_call_part(Trait tr, SimpleNode expr, Port pid, AssignmentVector asgn_vect)** Make a function call to Part object and return the result.

Class ObjectModelSupport: To support the evaluation of logical assertions included in the transitions of the specification of TROM classes. Its functions will invoke functions of LSLLibrarySupport.

Attributes	
Subsystem	ss
Tromclasslist	trom_asts
SCSlist	Scs_asts

LSLLibrarySupport	lls
Port	pid
AssignmentVector	asgn_vector
Methods	
	ObjectModelSupport(Subsystem subsys, Tromclasslist trom_ast_list, SCSlist scs_ast_list, LSLLibrarySupport lsl_lsup) Constructor.
boolean	**evaluate_assertion(SimpleNode expr, attributelist al, Port p_id , AssignmentVector asgn_vect)** Invoke evaluate_bool_expr(expr.get_children(), al)
boolean	**evaluate_bool_expr(SimpleNode expr, attributelist al)** Assert whether it is unary or binary bool expression. It will invoke evaluate_binary_bool_expr(expr, al) for binary bool expression and evaluate_unary_bool_expr(expr, al) for unary bool expression
boolean	**evaluate_binary_bool_expr(SimpleNode expr, attributelist al)** Evaluate binary bool expression.
boolean	**evaluate_unary_bool_expr(SimpleNode expr, attributelist al)** Evaluate unary bool expression.
boolean	**evaluate_inequality_expr(SimpleNode expr, attributelist al)** Evaluate inequality expression.
boolean	**evaluate_equality_expr(SimpleNode expr, attributelist al)** Evaluate equality expression.
int	**evaluate_unary_int_expr(SimpleNode expr, attributelist al)** Evaluate unary integer expression such as len(x).
Port	**evaluate_unary_port_expr(SimpleNode expr, attributelist al)** Evaluate unary port expression.
Trait	**evaluate_unary_trait_expr(SimpleNode expr, attributelist al)** Evaluate unary trait expression.

Class PortLink: To model the link between two objects defined in Configure section of SCS file.

Attributes	
TromPortTuple	tp_tuple1
TromPortTuple	tp_tuple2
Methods	
	PortLink(Trom t1, Port p1, Trom t2, Port p2) To establish the link between (t1, p1) <-> (t2, p2)

Class PortLinkList: List of PortLinkList objects.

Attributes	
TromPortTuple	tp_tuple1
TromPortTuple	tp_tuple2
Methods	
	PortLinkList() Default constructor
	PortLinkList(PortLink pl) Constructor
Parent Classes	
List	As the collection of PortLink objects.

Class Ports: To implement Round-bin algorithm.

Attributes	
int	port_card
String	port_typ
SimPortlist	port_lst
Node	lru_port_node
int	ports_tried
Methods	
int	**trial_count()** Get *ports_tried.*
void	**init_trial_count()** Initialize the variable *ports_tried* to 0.
void	**increment_trial_count()** Increment *ports_tried*
Node	**lru_port()** Get the least recently used port in all usable link ports for a given port according to the System Configuration Specification – SCS file.
Port	**get_port(String p_id)** Get the port that names *p_id.*

Class ReactionWindow: To represent a timing requirement in a time constraint. It is described by the lower and upper bound of time constraint as well as the corresponding SimulationEvent object.

Attributes	
int	low
int	up
SimulationEvent	se
Methods	
	ReactionWindow() Default constructor
	ReactionWindow(int l, int u) Constructor
	ReactionWindow(int l, int u, SimulationEvent simev)
	Constructor
int	**lower()** Return the lower bound - *lower*
int	**upper()** Return the upper bound - *up*
SimulationEvent	**get_scheduled_reaction()** Return the SimulationEvent object – *se* constrained by the indicated time range.
boolean	**within_window(int t)** To determine whether *t* is in the range [low, up]
boolean	**opeq(ReactionWindow w)** To determine whether it is equal to the ReactionWindow object - *w*
void	**display(PrintStream out)**
void	**display()**

Class ReactionWindowManager: Manage the ReactionWindow when events and transitions issue.

Attributes	
Subsystem	ss
TimeManager	tm
EventScheduler	es
Methods	
void	**handle_transition(SimulationEvent se, trans_spec ts_ast)** To handle the transition relative to the SimulationEvent *se*.

	It fires outstanding reactions, disables reactions associated with a time constraint when the TROM enters a disabling state, or enables reactions time-constrained by a transition.
void	**fire_reactions(SimulationEvent se, time_constraint tc_ast)** Create a ReactionHistory object to store historical state and maintain the ReactionWindow list.
void	**disable_reactions(SimulationEvent se, time_constraint tc_ast)** Disable reactions associated with a constraint when the TROM enters a disabling state.
void	**enable_reaction(SimulationEvent se, time_constraint tc_ast)** Create a ReactionHistory object to store historical state and invoke schedule_enabled_event() to schedule events constrained by time constraints that involves this transition as the start time.

Class ReactionVector: To model the timing **behavior** of the Trom class defined in .trom file. It encapsulates all reactions of a trom through maintaining a ReactionSubVectorList object.

Attributes	
ReactionSubVectorList	rsvl
Methods	
	ReactionVector(TROMclass statics) Constructor. Create ReactionSubVector for each time constraint and add them into ReactionSubVector list.
ReactionWindowList	**get_reaction_windows(time_constraint tc_ast)** Traverse ***rsvl*** to find and return the ReactionWindowList object corresponding to the time constraint ***tc_ast***

Class ReactionSubVector: To encapsulate some reaction information extracted from time constraint for dynamically handling.

Attributes	
time_constraint	tc_ast
ReactionWindowList	rw_list
Methods	
time_constraint	**time_constrain()** Get encapsulated time constraint
ReactionWindowList	**reaction_windows()** Get the Reaction Window List corresponds to the time constraint

Class Simulator: Simulator is the core class of the simulator module. It really executes the simulating function.

Attributes	
Subsystem	ss;
SimulationEventList	sel
TimeManager	tm
SimulatorPane	simPane;
boolean	dbg_mode
Trom	temp_trom
ObjectModelSupport	oms
SubsystemModelSupport	sms
ConsistencyChecker	cc
instantiatelist	temp_il
Node	temp_nptr
Debugger	dbg
AxiomGenerator	ag
Reasoning_system	rs
EventHandler	eh
ReactionWindowManager	rwm
Tromclasslist	Trom_asts
SCSlist	Scs_asts
SCSSimEv	Se_asts
EventScheduler	es
String	value[]

Methods	
	Simulator () Default constructor. Initialize all data members to null.
	Simulator(Reasoning_system resy, Subsystem subsys, SimulationEventList selist, TimeManager tmgr, ObjectModelSupport objms, SubsystemModelSupport subsysms, ConsistencyChecker cchk, Debugger dbger, AxiomGenerator axgen, Tromclasslist trom_ast_list, SCSlist scs_ast_list, SCSSimEv simev_ast_list, SimulatorPane sPane) Initialize data members with input values.
void	**initialise_sim(String d, String c, int t)** It invoke the initialize_subsystem() of class SubsystemModelSupport to intilize all subsystems and their included subsystems with the information in the AST. It performs the Time Interval Comparison algorithm and schedules all unconstrained output events as well as

	unconstrained internal events from initial state.
void	**handle_sim_debug_mode()** Traverse the Simulation Event List and handle each event one by one, which is done by invoking handle_event(). At same time, schedule unconstrained internal events that go out the current state of each trom object. Finally, output the events and Trom states to the screen.
void	**run()** It invokes handle_sim_debug_mode() or handle_sim() to handle events in Debug mode or Non-debug mode.
void	**handle_sim()** It is same as handle_sim_debug_mode() other than doing so continuously.
Subsystem	**get_ss()** Get the subsytem
TimeManager	**get_tm()** Get the time manager
SimulationEventList	**get_sel()** Get the Simulation Event List
Debugger	**get_debugger()** Get the debugger
Boolean	**debugger_mode()** To determin if it is in the debug mode
void	**set_debugger_mode(String d)** Set debug mode according the value of String *d*
void	**resortTimeConstraint(Trom trom)** Perform the Time Interval Comparison algorithm to sort the time-constrained events based on their time interval.
void	**insertByTimeInterval(time_constraint timeConstraintNode, LinkedList targetList)** It is used by the resortTimeConstraint() to perform Time Interval Comparison algorithm.It inserts timeConstraintNode to targetList according to the order of their time interval.

Class SimulatorPane: SimulatorPane is the graphical interface of the simulator.

Attributes	
Int_Sim	sim
Methods	
	SimulatorPane(Mainwindow mw) To construct the simulator user interface.

Internal Classes	
StartAction	Action listen of button "Start". Fired to execute start() of class Int_Sim.

Class Subsystem: Model subsystem defined in the SCS.

Attributes	
String	system_name;
SubsystemList	include_list
TromList	trom_list
PortLinkList	portlink_list
Methods	

PortLinkList	**portlinks()**
	return portlink_list

TromPortTuple	**get_linked_tromport_tuple(TromPortTuple tpt)** To get the TromPortTuple object corresponding to *tpt*. This TromPortTuple object is searched in the PortLink objects that is in the *portlink_list*

Class SubsystemList: List of Subsystem objects.

Attributes	
Methods	
	SubsystemList() Default constructor
	SubsystemList (Subsytem s) Constructor
Parent Class	
List	As the collection of Subsystem objects.

Class trans_spec: To model the transition, including source state, destination state, condition, and trigger event etc.

Attributes	
String	transition_label
boolean	if_initial_transition
state	source_state
state	destination_state

event	triggering_event
Trans_Parmlist	parms_and_values
ASTStart	port_condition
ASTStart	enabling_condition
ASTStart	post_condition
Methods	
	trans_spec(String tl, boolean ins, state ss, state ds, event te, **Trans_Parmlist pv, ASTStart pc, ASTStart ec, ASTStart poc)** Constructor
	trans_spec() Default Constructor
void	**set_trans_label(String tl)** Set the name of the transition as *tl*.
String	**get_trans_label()** Get the transition name
boolean	**get_if_initial_state()** To determine whether this transition is going out from initial state.
void	**set_if_initial_state(boolean ins)** Set true if this transition is going out from initial state.
void	**set_source_state(state ss)** Set the source state as *ss*.
state	**get_source_state()** Get the source state
void	**set_destination_state(state ds)** Set the destination state as *ds*.
state	**get_destination_state()** Get the destination state.
void	**set_triggering_event(event te)** Set the triggering event as *te*.
event	**get_triggering_event()** Get the triggering event.
void	**set_parms_and_values(Trans_Parmlist pv)** Set the parameter expression list as *pv*.
Trans_Parmlist	**get_parms_and_values()** Get the parameter expression list
void	**set_port_condition(ASTStart pc)** Set the port condition as *pc*.
ASTStart	**get_port_condition()**

	Get the port condition.
void	**set_enabling_condition(ASTStart ec)** Set the enabling condition as *ec*.
ASTStart	**get_enabling_condition()** Get the enabling condition.
void	**set_post_condition(ASTStart pc)** Set the post condition as *pc*.
ASTStart	**get_post_condition()** Get the post condition.
void	**display(PrintStream out)** Display all information corresponding to this transition
void	**display_labels(PrintStream out)** Display all labels such as state name, event name etc.

Class Trom: To model the trom object defined in the SCS file, including its dynamic state and event list etc.

Attributes	
String	trom_label
String	class_label
PortsList	port_list_list
state	curr_stat
AssignmentVector	asgn_vect
ReactionVector	reac_vect
TROMclass	statics
SimulationEventList	history
LSLLibraryManager	llm
Methods	
Ports	**get_port_list(String port_type_name)** Traverse the port_list_list to find and return the Ports object whose port type name is same as the *port_type_name*; otherwise, return null.
String	**label()** Get Trom name - *trom_label*
String	**trom_class()** Get Class name - *class_label*
state	**current_state()** Get the current state of Trom object - *curr_stat*
TROMclass	**ast()**

	Get the TROMclass - *statics*

Class TROMclass: Model the TROM classes defined in .trom files.

Attributes	
String	class_name
portlist	port_type_list
eventlist	event_list
statelist	state_list
attributelist	att_list
LSLtraitlist	lsl_trait_list
att_funclist	att_func_list
trans_speclist	trans_spec_list
time_constraintlist	time_constraint_list
Methods	
boolean	**is_a_constrained_event(event e)** To determine whether an event is a constrained event. Input event is not a constrained event. It also searches time_constraint_list to find whether this event is encapsulated in this list.
eventlist	**get_unconstrained_internal_event(state s)** To find all events that is internal and not constrained and their states are same as *s*.
void	**set_classname(String s)** Set the TROM class name as *s*
String	**get_classname()** Get the TROM class name
void	**set_portlist(portlist pl)** Set the port list as *pl*. The port list is the list following the TROM class name in the definition of the TROM class.
portlist	**get_portlist()** Get the port list
void	**set_eventlist(eventlist el)** Set the event list as *el*. The event list corresponds to the list defined in the Events section of the TROM class.
eventlist	**get_eventlist()** Get the eventlist
void	**set_statelist(statelist sl)** Set the state list as *sl*. The state list corresponds to the list defined in the States section of the TROM class.

statelist	**get_statelist()** Get the statelist
void	**set_attribute_list(attributelist al)** Set the attribute list as *al*. The attribute list corresponds to the list defined in the Attributes section of the TROM class.
attributelist	**get_attributelist()** Get the attribute lsit
void	**set_LSLtraitlist(LSLtraitlist lsl)** Set the LSL trait list as *lsl*. The LSL trait list is the list of trait defined in the Traits section of the TROM class.
LSLtraitlist	**get_LSLtraitlist()** Set the LSL trait list
void	**set_att_func_list(att_funclist afl)** Set the attribute fuction list as *afl*. The attribute fuction list is the list defined in the Attribute-Function section of the TROM class.
att_funclist	**get_att_func_list()** Get the attribute fuction list
void	**set_trans_speclist(trans_speclist tsl)** Set the transition specification list as *tsl*. The transition specification list corresponds to the transition list defined in the Transition-Specifications section of the TROM class.
trans_speclist	**get_trans_speclist()** Get the transition specification list
void	**set_time_constraintlist(time_constraintlist tcl)** Set the time constraint list as *tcl*. The time constraint list corresponds to the time variable list defined in the Time-Constraints section of the TROM class.
time_constraintlist	**get_time_constraintlist()** Get the time constraint list

Class TromPortTuple: To model the link between objects such as (t1, p1).

Attributes	
Trom	tpt_trom
Port	tpt_port
Methods	
	TromPortTuple() Default constructor
	TromPortTuple(Trom trom, Port port) Constructor

boolean	**opeq(TromPortTuple tpt)** To determine whether two TromPortTuple objects are same.