A FORMAL VERIFICATION ASSISTANT FOR TROMLAB ENVIRONMENT

FRANÇOIS POMPEO

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

SEPTEMBER 1999

0-612-43667-5

Abstract

A Formal Verification Assistant for TROMLAB Environment

François Pompeo

Formal specifications have become a strong basis in the field of safety critical systems development. Safety, liveness and time bounded properties are characteristics of such systems where the need to secure their adequate implementation is very high. Formal verification of such properties is the research field of this thesis. It presents an automated tool that enables mechanized axiom extraction from real-time reactive systems. It is implemented within TROMLAB which is a development environment based on the Timed Reactive Object Model (TROM). The objective of this tool is to be used within the verification methodology of TROM as an automated assistant to facilitate time dependent property proving for model developers.

Acknowledgments

I would like to deeply thank my supervisor, Dr. Alagar. I truly believe that the implication and devotion demonstrated to his students and research make him part of the elite class of professors that all students should hope for. I thank Darma Muthiayen for his insightful discussions.

Je remercie mes parents pour m'avoir donné tout les moyens possibles pour compléter ce but qui m'étais si important.

Merci maman, grazie papa.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Real-time reactive systems

Reactive systems have as a characterizing feature the ability to be in continuous interaction with their environment. Their behavior obeys a stimulus and response mode of conduct. Hardware interfaces act for contact to the environment as devices that react to physical stimulus and as devices that influence or modify the environment. Examples of such systems are alarm systems, nuclear reactor control systems, air traffic control systems or telecommunication systems to name a few. When we add the real-time aspect, time regulation is introduced. In other words, real-time systems are regulated by time constraints formulated in the design of the system. Real-time reactive systems are therefore systems that are in constant relationship with their environment and the stimulus-response behavior respects time constraints that ensure its correct and safe operations.

Correctness and safety are inherent goals of the design of real-time reactive systems as safety critical contexts are most often the environments where these systems operate. The examples stated above demonstrate this. The analysis of the functional and timing properties must be as exhaustive as possible. Because the failure of the real-time reactive systems may have catastrophic consequences, the whole software community, customers as well as development teams need compelling evidence that such systems deliver functionality and timing properties as desired [HD96]. To obtain this evidence, people are willing to invest considerable time, effort and money.

Formal methods are currently a well studied avenue to answer this need. Formal specifications and methods allow demonstration of a system's ability to uphold critical properties.

Foundational work has been laid on formal reactive system modeling [Ach95]. The *Timed Reactive Object Model* (TROM) formalism and the notion of abstract (generic) reactive model (GRC) are introduced. The complete semantics of the formalism, and several case studies demonstrating the expressiveness of the formalism are shown in [Ach95] and [AAM96]. TROMLAB [AAM96] is a development environment for real-time reactive systems based on the TROM formalism. An overall architectural view of TROMLAB can be seen in Figure 1.

The following components of the TROMLAB environment are currently operational:

- **Rose-UML translator** - [Pop99] A translator to extract TROM specifications from Rose-UML based on [AM98] UML extensions;
- **Graphical User Interface** - [Sri99] A graphical front-end modeling and interaction facility to the TROMLAB environment;
- **Interpreter** - [Tao96] A parser, syntax checker and internal representation builder, the Abstract Syntax Tree (AST);
- **Simulator** - [Mut96] A subsystem animation tool based on the AST and validation tool;
- **Browser** - [Nag99] A library browser for navigation, query and access to system components.
- **Reasoning System** - [Hai99] A system debugging tool to be used during animation by facilitating interactive queries of hypothetical nature on system behavior.

1.2 Formal verification

It has been identified that the principal advantages to formal methods in designing real-time reactive systems as being the support for: verification of desired system properties through application of formal proving techniques; verification of the correctness of specifications through model simulations; production of preliminary implementation code; and test suite generation for implementation checking [Bol96]. It is said that formal specifications have as main feature the support of formal deduction, in other words, the possibility to reduce certain questions to a process closely resembling calculations that can be checked by others and by machines [COR$^+$95]. Finally and by definition formal methods refer to the use of concepts and techniques from logic and discrete mathematics.

All of these items are of interesting importance but the focus of the work included in this thesis relates to verification. The goal of verification as a general term can be seen by two definitions. The first one being the action of insuring that the behavior of the implementation is what was intended. The second one is to verify not only that specifications are respected but to *validate* the specifications themselves. The specifications must be complete, consistent, capture the stated needs and finally satisfy *critical properties*. Such properties are usually categorized into safety properties, liveness properties, and bounded-time properties.

Model-Theoretic reasoning - Many approaches to state machine model verification have been developed. One class of algorithm [CES86], *model checkers* was successfully used for untimed specifications verification. The algorithms based on *model checking* take a finite state machine model of a system and temporal logic formulas and determine if the formulas are true for the model. The application of *model checking* to timed specifications remains a difficult goal since adding time to the specifications often produces models that are too large to analyze.

Proof-theoretic reasoning - In proof-theoretic reasoning, a theory is developed about the system in some logic, such as higher-order logic. The system properties are then expressed as theorems to be validated against the theories. Although developing proofs can be costly in time and efforts, there are some advantages to the proof-theoretic reasoning approach.

- Better model abstraction can lead to more generalized results. For example, in state machine modeling, reasoning can can be on an infinite number of states and variables can be used for timing constraints (as opposed to finite number of states and constants)
- By developing proofs, designers gain a deeper understanding of the specifications and its properties, such as the dependencies and boundary conditions
- State machine is not the only available model, proof-theoretic techniques can be applied to any mathematical model.

This thesis is developed in the context of the TROM formal model, hence having time properties. Moreover, the goal is to specifically validate timed properties. Hence, proof-theoretic reasoning is the selected approach. PVS [ORS92] is the selected mechanical proof system of our research team, it uses higher-order logic. Other systems exist, such as Larch Prover [GH93] or Boyer-More prover [BM88], they use first-order logic. By selecting higher-ordered logic models decrease in complexity and increase in generalizing power. By using mechanized proof systems, one can increase the confidence in the proof's validity, especially in safety-critical contexts. A more detailed description of PVS will be given in Chapter 3.

The current status of the research community regarding proof asisting is fairly developed. Many provers such as Larch, PVS and others do have capabilities to perform formal proving. The challenges of this work lie in the axiomatization of state machine based formal specifications to a formal proving environment. After consulting the literature within this specific field, it was found that no other tool currently tackles this problem.

1.3 Research goals

With the methodology introduced in [MA99] as grounds for the work of this thesis, the objective of my research work was to apply and in some cases refine the methodology to enable a clear derivation of an axiomatic description of the formal specifications to apply proof-theoretic reasoning. Moreover, a tool that has for foundation this methodology, was developed to help real-time reactive system designers in their formal proving process of safety properties of the model at hand.

Therefore here are the main contributions of this thesis:

1. Refining discussions on axiomatic descriptions of the TROM formalism.

2. The development of a tool within the TROMLAB environment for an automated axiom derivation based the methodology described in [MA99]

3. An application of the TROM formalism to a Robotic Assembly System with an application of the axiomatic description methodology with an automated output.

As described in [MA99], the significance of the axiomatic derivation from the TROM model is the use of PVS [ORS92] as back-end for mechanized verification. Moreover, the TROMLAB environment now has a graphical front-end with the recently integrated ROSE-UML translator from [Pop99]. With the addition, as middle-ware, of the TROM-axiomatic description generator described in this thesis, we see the beginning of a fully mechanized specification life cycle starting with graphical input all the way to mechanized formal proving going through the animation/validation of the models.

The structure of this thesis is as follows. Chapter 2 presents the formalism of TROM and presents the notions of GRCs (Generic Reactive Class). Chapter 3 presents all the needed ingredients for formal proving of safety properties with axiomatic description through PVS proof mechanization and the *since* operator. Chapter 4 describes the design details of the mechanized axiomatic generation tool. Chapter 5 presents a case study using a rather complex example that demonstrates the usefulness of the tool. This example is a Robotic Assembly System first introduced in [AAR95a]. The thesis ends with Chapter 6 which presents the conclusions to be drawn from this thesis work and also presents the future work angle.

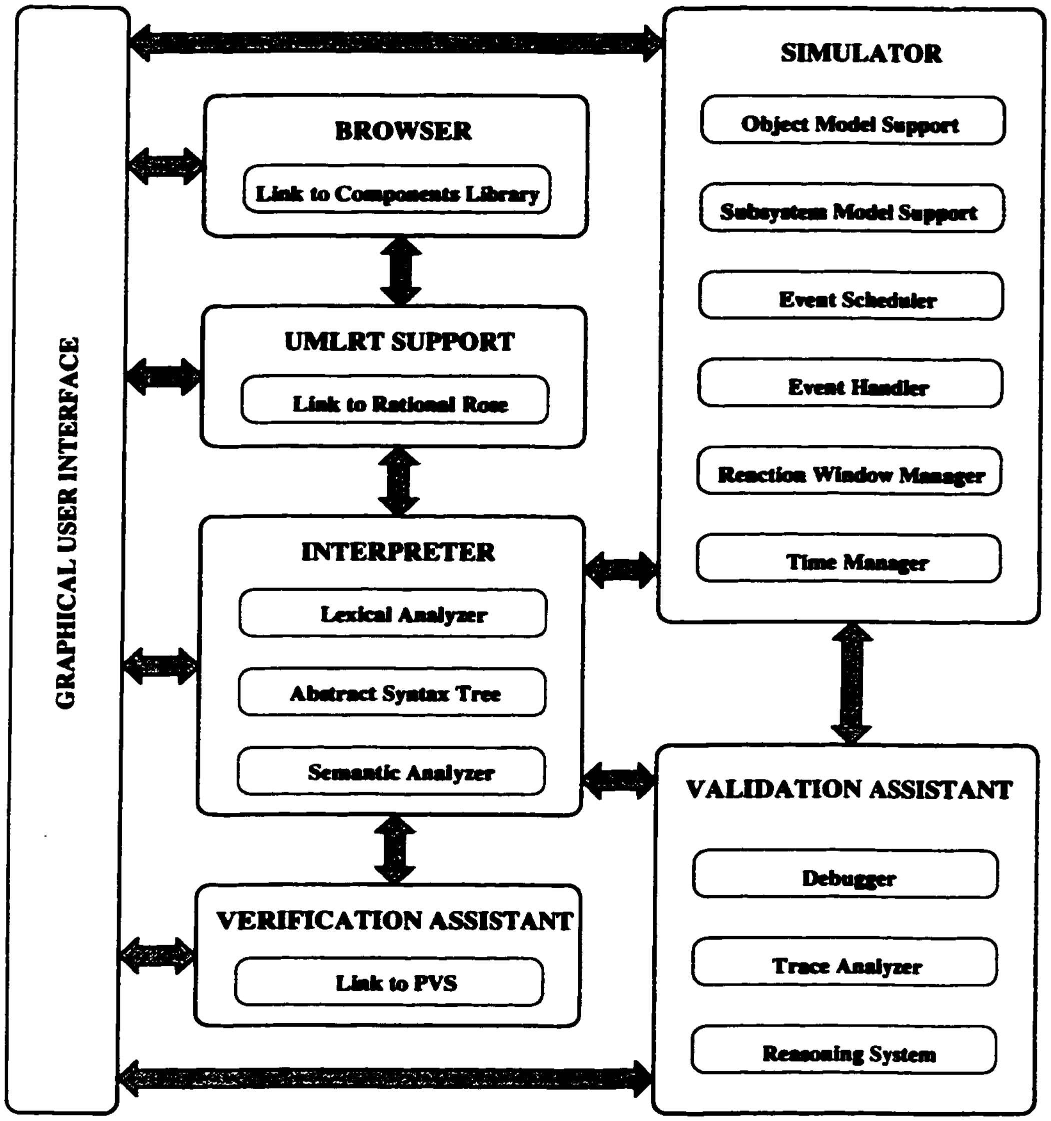


Figure 1: Existing TROMLAB architecture

Chapter 2

The GRC formalism

2.1 Introduction

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

An object-oriented modeling technique for real-time reactive systems was introduced in [Ach95]. It introduces the *Timed Reactive Object Model* formalism, and the notion of an abstract (generic) reactive model. A complete semantics of the formalism, and several case studies illustrating the expressiveness of the formalism have appeared in [Ach95, AAR95b].

2.2 The informal model

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

Informally, a reactive object consists of the following elements:

- **A set of events** partitioned into internal, input and output events. *Input and Output events* occur at a *port* and represent message passing. The names of these events are suffixed by ? and !, respectively. *Internal events* are assumed to occur at the *null* port.

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

Figure 2 illustrates the elements of a reactive object.

2.3 The formal model

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

A reactive object is an 8-tuple ($\mathcal{P}$, $\mathcal{E}$, Θ, $\mathcal{X}$, $\mathcal{L}$, Φ, Λ, Υ) such that:

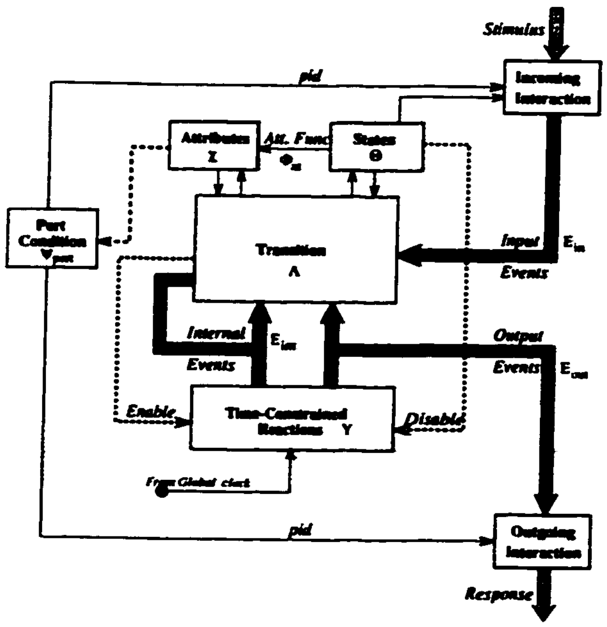


Figure 2: Anatomy of a reactive object

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3: Template for System Configuration Specification.

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

Figure 4: Template for System Configuration Specification.

2.4 The TROM logical semantics

This section introduces the semantics of the **TROM** model expressed through a set of axioms that are called the logical semantics. This logical semantics is used for two main purposes. First as a set of rules to check the well-formedness of a **TROM** model, and second as ground for the formal verification methodology. The currently used logical semantics were originally described by [Ach95] and later adapted by [AM99] with the OCL (Object Constraint Language) to comply to their UML TROM definitions. The complete description of all axioms can be found in [AM99]. Therefore only a subset with the relevant axioms for this thesis will be described in details in the next Section 2.4.1, which are the ***transition axiom*** (9) which describes the effect of an event within an object, the ***constrained event axiom***(11b) which describes the upper and lower time limit of a constrained event firing delay and the ***synchrony axiom*** (12) which describes the synchronous message passing between linked objects.

In order to support a semantic definition of the logical assertions, three OCL domains are introduced. A reactive object domain, a reactive subsystem domain and a domain for time intervals. All of these are used in the definition of the predicates on time intervals to assert time-dependent properties on elements from the domain of reactive objects, that is the TROM logical semantics axioms. Here are the predicates for the time interval domain:

- $HoldAt(s,t)$ which asserts that an object is in state s at time t.
- $HoldDuring(s,T)$ which asserts that an object holds state s for the time interval T. The $HoldDuring$ can be defined with $HoldAt$ with the following. If time interval $T=[u,v]$ then for an object A

$$A.HoldDuring(s,T)\ \textit{implies}\ \forall t:\ u \le t \le v\ \textit{implies}\ A.HoldAt(s,t).$$

- $Occur(e,p,t)$ which asserts that event e occurs at port p at time t.

With these predicates defined, the logical semantics used as basis for our derivation algorithms can be stated.

2.4.1 Axiom system

There are eleven axioms of temporal constraints associated with an instance of a generic reactive class. The *Synchrony* axiom describes the semantics for synchronous message passing. An OCL expression of the form

$$\text{self}.events\text{->forall}(e \mid P(e))$$

applied to a reactive object, denotes *"for all events e of the GRC instance, predicate P is true"*. The variable t for the time of an event occurrence denotes a discrete time point.

1. *Atomic-event axiom:* .... **(AE)**

   At time t, there can be at most one event occurring in a reactive object; at time t, an event can occur at only one port.

2. *Silent-event axiom:* .... **(SE)**

   The occurrence of the silent event *tick* at time t precludes the occurrence of any other event in the reactive object at time t.

3. *State-hierarchy axioms:* .... **(SH)**

   These axioms assert the relationship between a state and its sub-states. When an object is in a sub-state of a state θ, it is also in the state θ. Similarly, when a reactive object is in a non-atomic state θ, it is in at least one of the sub-states of θ.

4. *State-uniqueness axiom:* .... **(SU)**

   A reactive object cannot be in more than one state at any instant, unless the states are related by the state hierarchy function Φ_s. That is, a reactive object can be in two states only if one state is a subs-tate of the other. Formally,

5. *Initial-state axiom:* .... **(IS)**

   A reactive object has a unique initial state which is atomic. A reactive object is in its initial state θ_0 at the initial instant t_{init}.

6. *Initial-attribute axiom:* .... **(IA)**

   A formula φ_{init} is asserted at the initial time t_{init} such that φ_{init} is the maximal property satisfied by the attributes at t_{init}. The assertion φ_{init} is the maximal property in the sense that, for any other assertion φ satisfied by the attributes at time t_{init}, the following holds:

7. *Dormant-attribute axiom:* .... **(DA)**

   The attribute function partitions the attribute set into modifiable and non-modifiable sets, at each state. If an attribute is dormant in a certain state then its value cannot be changed as long as the machine is in that state.

8. *Occurrence axiom:* .... **(OC)**

   For the occurrence of signal $Occur(e, p_i, t)$ it is necessary that the reactive object be in the source-state of some transition λ, labeled by e, such that the port-condition φ_{port} of λ is satisfied by p_i. This is formalized by the occurrence axiom asserted for each event e in the reactive object. For an event e, let $\lambda_1, \ldots, \lambda_n$ be the transition specifications labeled by e, and let θ_j be the source-state of λ_j, φ^j_{en} be the enabling-condition of λ_j and φ^j_{port} be the port-condition of λ_j. The occurrence axiom for e follows.

9. *Transition axiom:* .... **(TR)**

   The transition axiom is defined for each transition specification of a reactive object. The occurrence of an event results in a state transition to the target state and the satisfaction of the post-condition in the target state. The transition axiom applies for each transition specification $\lambda : \langle\theta, \theta'\rangle; e(\varphi_{port}); \varphi_{en} \Longrightarrow \varphi_{post}$. If the target state θ' is not an atomic state, then the atomic state which is the starting descendant state of θ' replaces θ'.

```
self.$HoldAt(s_1, t_1)$ and self.$Occur(e, p, t_1)$
  and $t_1 < t_2$ implies
    self.$transitions$->exists($r$ | $r.source = s_1$
      and $r.destination = s_2$ and self.$HoldAt(s_2, t_2)$
        and $r.postcondition(t_1, t_2, p)$ = true)
```

10. *Persistence axiom:* .... **(PS)**

    A persistent axiom is defined for each state. It asserts that when no event causing a transition to leave that state occurs, there is neither a change in that state nor a change in the values of the attributes active in that state. For a state θ, let $e_1, \ldots, e_n$ denote the events associated with the transitions leaving θ. The persistent axiom for θ is as follows.

11. *Time-constraint axioms:* .... **(TC)**

    A set of time constraint axioms defines the behavior of a reactive object. The axioms apply for each time-constraint

$$(\lambda, e, [l, u], \Theta_i) = \upsilon_i \in \Upsilon,$$

where $\lambda : \langle\theta, \theta'\rangle; f(\varphi_{port}); \varphi_{en} \longrightarrow \varphi_{port}$. We introduce the predicates *Enable*, *Disable*, and *Trigger*, to describe the status of a reaction after it has been enabled, and the predicate *Within* to assert the containment of a time point within a bounded time interval.

- $Trigger(e, t_a)$: A reaction is activated when a transition triggering the reaction occurs. For a time-constraint υ_i, the occurrence of a trigger transition λ is marked by a change of state from θ to θ' and the occurrence of the labeling event f. $Trigger(e, t_a)$ is true when a reaction associated with the constrained event e is activated at time t_a. If e is not a constrained event then $\forall t$, $\neg Trigger(e, t)$ is true.

```
$Trigger(e, t_a) \overset{\text{def}}{=}$ self.$transitions$->exists($r$ |
  $r.triggerevent = f$ and self.$Occur(f, p_i, t_a)$
    and self.$HoldAt(r.source, t_1)$
      and self.$HoldAt(r.destination, t_2)$
        and $t_1 < t_a$ and $t_a < t_2$
          and self.$timeconstraints$->exists($tc$ |
            $tc.assoctransition = r$
              and $tc.constrainedevent = e$))
```

- $Disable(e, t)$: Any activated reaction involving the constrained event e is disabled at time t due to the reactive object entering one of the disabling states of e. If e is not a constrained event then $\forall t$, $\neg Disable(e, t)$ is true.

```
$Disable(e, t) \overset{\text{def}}{=}$ self.$timeconstraints$->exists($tc$ |
  $tc.constrainedevent = e$ and
    $t < tc.upperbound$
      and $tc.disablingstates$->exists($s$ | self.$HoldAt(s, t)$))
```

- *Enable*(e, t_a, t): The reaction involving the constrained event e due to the occurrence of a trigger event at the activation instance t_a is enabled at time t. An event e is enabled at time t if it was triggered at time t_a, $t_a < t$, and it was not disabled or fired at any time t', $t_a < t' \leq t$. A formal definition of the predicate follows from the axioms stated below. If e is not a constrained event then $\forall t_a, t,\ \neg Enable(e, t_a, t)$ is true.
- We define the predicate $Within(t_a, l, u, t)$ in terms of the basic temporal predicates.

$$Within(t_a, l, u, t) \overset{\text{def}}{=} t_a + l \leq t \leq t_a + u$$

The following axioms use the predicates $Trigger(e, t_a)$, $Disable(e, t)$, and $Enable(e, t_a, t)$, and the temporal predicates to describe the behavior of objects of the generic reactive classes.

(a) *Activation axiom:* .... **(ac)**

A reaction is activated when a transition triggering the reaction occurs.

(b) *Constrained-event axiom:* .... **(ce)**

A trigger event is necessary for the occurrence of a constrained event.

```
self.$timeconstraints$->forall($tc$ | $tc.constrainedevent = e_1$
  and self.$Occur(e_1, p_i, t)$ implies
    self.$transitions$->exists($r$ | $r.triggerevent = e_2$
      and self.$Occur(e_2, p_j, t_a)$
        and $Within(t_a, l, u, t)$
```

(c) *Enabling axiom:* .... **(en)**

The necessary conditions for a reaction already enabled at time t to remain enabled in the succeeding time t' are: (1) the constrained event e should not occur at t, and (2) the reaction is not disabled at time t'.

(d) *Disabling axiom:* .... **(ds)**

An enabled reaction will no longer be enabled if the constrained event of the reaction is disabled due to the object entering into a disabling state.

(e) *Firing axiom:* .... **(fr)**

An enabled reaction is fired by the occurrence of the constrained event. Since the firing of the reaction satisfies an enabled reaction, the reaction will no longer be enabled.

(f) *Prohibition axiom:* .... **(ph)**

If a reaction is enabled then the constrained event should not occur during the minimum delay period from the time of activation. However, if the minimum delay is less than the atomic interval, then there does not exist any minimum delay interval.

(g) *Obligation axiom:* .... **(ob)**

If an enabled reaction is not disabled within the maximum time bound after the activation, then the constrained event should be fired at some time within the maximum time bound.

(h) *Validity axiom:* .... **(va)**

A reaction involving a constrained event e can be enabled at time t only if the triggering event f has occurred at time t_a such that t is within the maximum bound u from the instant t_a. In other words, for a constrained event activated at a given time t_a, for all time instants t', such that, $t' < t_a$ or $t' > t_a + u$, the constrained event e cannot be enabled. By including this axiom, we can assert whether or not the predicate $Enable(e, t_a, t)$ is true for all constrained events e and time instants t_a and t.

12. *Synchrony axiom:* .... **(SY)**

The synchrony axiom applies for each port-link $o_i.@q_j \leftrightarrow o_k.@q_l$ in a *Subsystem Configuration Specification.*

```
self.$portlinks$->forall($pl$ | self.$instances$->exists($o_1, o_2$ |
  $pl.instance_1 = o_1$  and  $pl.instance_2 = o_2$
    and  $(o_1.Occur(e, pl.port_1, t)$
      implies  $o_2.Occur(e, pl.port_2, t))$
    and  $(o_2.Occur(e, pl.port_2, t)$
      implies  $o_1.Occur(e, pl.port_1, t)))$
```

Chapter 3

GRC verification

3.1 Verification process

In order to understand the motivations behind the axiom generation of the TROM specifications, we need to present the context in which these axioms are used. The general context is the TROMLAB environment, which already has tools to handle the TROM specifications.

The goal as stated in the introduction of this thesis, is to obtain formal certification of a model's ability to fulfill its required properties. In Figure 5 it is the goal identified as A. The process a model designer would go through to achieve this goal is the following, again referring to Figure 5:

1. User defines the problem at hand and uses one of the three available methods to enter the formal TROM specifications, UML model [Pop99], formal specifications [Ach95] or the TROMLAB GUI [Sri99]. All of these methods lead to the TROM semantics.

2. The formal specifications can be parsed and passed through a semantics checker to obtain the abstract syntax tree (AST) which is the internal representation of TROM specifications [Tao96, Sri99].

3. Users can then use the tool developed as part of this thesis work to obtain one of the sets of axioms, either *since* expressions or PVS.

4. Users have defined their safety properties that eventually get to be expressed in PVS.

5. Proving of safety property within theorem prover can then be attempted.

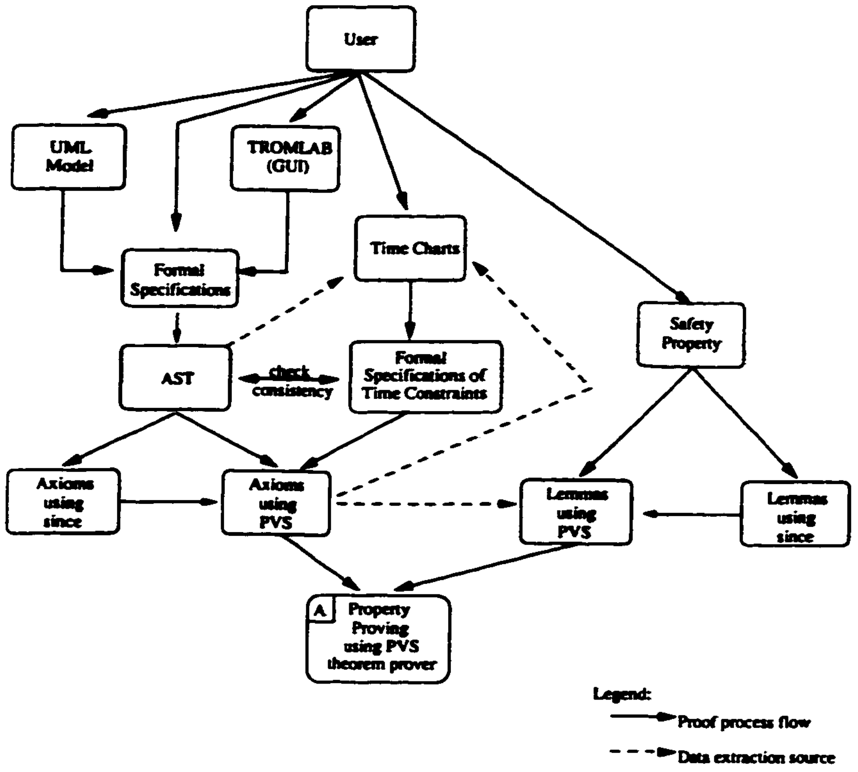


Figure 5: TROMLAB proof process

3.2 About PVS

PVS is a prototype system within an interactive environment for writing formal specifications and constructing proofs [ORS92]. It provides an expressive specification language based on higher-order logic, augmented with a typing system, parameterized theories and mechanisms to enable definitions of abstract data types such as lists and trees. Standard types defined in PVS include numbers, records, arrays, functions, sets and many more. The typing system therefore enables type checking for the specifications at hand and detect many basic specification errors very early. A detailed description and tutorial for the PVS specification language can be found in [AM, COR$^+$95].

Coupled with this specification language, an interactive theorem prover, referred to as a proof-checker, exists is PVS. The high-level functional descriptions of a system can have its desired properties proved. For example if a function that reverses a list has been correctly specified, a proof that the original list is obtained after reversing it twice can be built. Hence, absolute confidence about the function's correctness is obtained. One can also consider an automated proof assistant such as PVS as a skeptic rejecting any arguments that are not watertight. Hence requiring specification refinement or corrections.

The use that the TROMLAB team has for PVS is not a tool for full prototype specification and then proving properties. It is to use PVS as a back-end system for proofs of time dependent properties of TROM formalism based specifications. In other words, TROM specifications are to be transformed in order to use the PVS theorem prover. In the following sections, the transformations needed to go from TROM to the PVS environment will be shown first in a theoretic fashion and in Chapter 4 the algorithms used in the mechanized transformation tool is explained.

3.3 PVS model of TROMs

In [MA99] the following concepts are introduced as grounds for an axiomatic description of design specifications. The *computation* of an object is a general sequence of state transitions for an object. This corresponds to a series of events synchronized with these transitions. This correspondence is defined as the *duality* of event occurrences and state transitions. These definitions being in the context of reactive systems, an object's computation can be infinite due to the nature of reactive systems which involves constant interaction with its environment. Therefore, the concept of a *period* is also introduced which is a segment of the sequence constituting the computation of an object, moreover such a segment starts and ends with the object in its initial state. Such a segment cannot have the initial state within. Also, different periods within the computation of an object do not necessarily imply an identical sequence of events, different periods may mean different transitional paths or event sequences. A period can also include multiple occurrences of an event due to a cycle within a period. Hence the need for an event *occurrence* concept.

Axiomatic description is the basis for the verification methodology of [MA99]. From the TROM reactive system design, three types of axioms can be derived. *Transition* axioms, *time-constraint* axioms and *synchrony* axioms. A fourth type of axioms *supplementary* axioms need to be defined but are derived from the specifications, they are inherent properties to the model that do not appear explicitly in the design. The combined set of axioms creates a specification set of time properties of the reactive system against which time related safety properties can be proved. Transition axioms specify the ordering relation of an object's state transitions, time-constraint axioms specify time constraints on reactions to transitions and synchrony axioms specify the synchronization of message exchange between objects of a system. Finally the supplementary axioms are, as stated above, added to bring specific properties of the modeled system.

The basis of the PVS axiomatic description is the use of a higher-order function for every GRC of a system that expresses absolute time. This time function corresponds to the *occurrence* of an *event* within a *period* of a *GRC instance*. This function is defined as follow.

```
TT:[GRC -> [Period -> [GRC_event -> [occurrence -> Time]]]]
```

GRC is the set of instances of a generic reactive class, GRC_Event is the set of possible events within the specific GRC.

Transition axioms and time-constraint axioms are GRC specific. That is, a set of both type of axioms is to be developed for each GRC. Synchrony axioms are subsystem specific and therefore such axioms are developed once per defined subsystem. As for the supplementary axioms, such axioms do not obey specific rules for their development since they are problem type specific. Such axioms could be needed with each GRC or in the subsystem or within specific GRCs and so on.

The precise definition of these types of axioms are given in [MA99] therefore the next paragraphs will show a short theoretical derivation rules for the transition, time-constraint and synchrony axioms. All of these will be treated in greater detail in Chapter 4.

Transition axioms - Starting from initial state, if the destination state of a transition R_j is the same as the source state of a transition R_k, an axiom stating that the occurrence of event e_1 triggering the transition R_j precedes the occurrence of event e_2 triggering the transition time of R_k is to be included.

$$TT(A)(i)(e_1)(j) < TT(A)(i)(e_2)(j)$$

Time-constraint axioms - The time interval during which a reaction (event) to a transition (event) is constrained to occur is to be considered. Therefore, we include axioms stating that the delay between the occurrence of an activation event e_1 and the reaction event e_2 is to be greater than the lower bound and less than the upper bound expressed in the time constraints of the specifications.

$$TT(A)(i)(e_2)(j) - TT(A)(i)(e_1)(j) > l \text{ and}$$
$$TT(A)(i)(e_2)(j) - TT(A)(i)(e_1)(j) < u$$

where $[l, u]$ is the allowed window for the reaction event.

Synchrony axioms - A synchrony axiom is included for *port-links* defined in subsystems' configurations. Such links express the fact that for an event e occurring in object A_1 the same event e occurs in object A_2. Hence the occurrence time is equal.

$$TT(A_1)(i)(e)(j) = TT(A_2)(i)(e)(j)$$

The PVS specifications for the axiomatic description of the TROM specifications are structured in terms of theories. Two standard theories not related to the GRCs define some ground concepts. First the theory *model* defines the *Time* domain as the set of non-negative reals, the *Period* and *Occurrence* as the set of positive naturals. Secondly, the theory *transition_time* defines the function *TT* as seen earlier. Then a theory is developed for each GRC of the modeled system. These theories include the following:

- An uninterpreted non empty type for the instances of GRCs
- An enumerated type defining the set of allowable events in the GRC
- Declaration of universally quantified logical variables for *Period*, *Occurrence* and the class of objects
- The set of transition axioms
- The set of time-constraint axioms

For each subsystem definition a theory is defined with the following:

- An importing clause to include the GRC defining theories
- An importing clause to include the other defined subsystems
- Declaration of universally quantified logical variables for *Period* and *Occurrence*
- Declaration of universally quantified logical variables for the classes of objects in the case of general proofs
- Declaration of constants corresponding to the objects instantiated in the subsystem
- The set of synchrony axioms
- The set of supplementary axioms

We will see in Chapter 4 the refinement of algorithms to extract from the TROM specifications the axioms to be generated in the PVS theories. In Chapter 5 the reasoning behind every axiom generation within a case study with the generated PVS theories will be given.

3.4 About *since*

[Sha92] has introduced the *since* expression as a duration measure for real-time behavior reasoning. This measure expresses the time elapsed since a predicate was last true. In other words, if predicate P becomes false at time t, the value of *since*(P) (also written $|P|$) at time $t + x$ is $(t + x) - t$, which is x. [CM92] have introduced earlier a similar operator, ***punch*** which also operates on assertions but records the absolute time at which the assertion last went from false to true. Figure 6 shows predicates P and Q and their conjunction an disjunction over time. The times t_p, t_q are the absolute times at which the predicates P and Q were last true. $t_{p \wedge q}$ and $t_{p \vee q}$ are the absolute times at which the conjunction and disjunction of P and Q were last true. Table 1 shows the properties of the *since* operator according to the lemmas introduced by [Sha92]. The lemmas capture invariants on the behavior of the *since* operator.

With this relationship between the *since* operator and absolute time we are able to establish a pattern to derive linear equalities, in our case the higher-ordered PVS time expressions.

Table 1: Properties of the *since* operator.

Using *since* operator	Using absolute times
$\{\lvert(\lvert P\rvert \le x)\rvert \le y \supset \lvert P\rvert \le x+y\}$	$t_{\lvert(\lvert P\rvert \le x)\rvert \le y} = t_{\lvert P\rvert \le x} + y$ $= t_{\lvert P\rvert} + x + y$
$\{\lvert P \vee Q\rvert \le \min(\lvert P\rvert, \lvert Q\rvert)\}$	$t_{P\vee Q} = \max(t_P,\ t_Q)$
$\{\lvert P \vee Q\rvert = \lvert P\rvert \ \vee\ \lvert P \vee Q\rvert = \lvert Q\rvert\}$	$t_{P\vee Q} = t_P \ \vee\ t_{P\vee Q} = t_Q$
$\{\lvert P \wedge Q\rvert = \lvert P\rvert \ \vee\ \lvert P \wedge \neg Q\rvert = \lvert P\rvert\}$	$t_{P\wedge Q} = t_P \ \vee\ t_{P\wedge\neg Q} = t_P$
$\{\max(\lvert P\rvert, \lvert Q\rvert) \le \lvert P \wedge Q\rvert\}$	$t_{P\wedge Q} = \min(t_P,\ t_Q)$
$\{\lvert P\rvert \le \lvert P \wedge Q\rvert\}$	$t_P \ge t_{P\wedge Q}$
$\{\lvert Q\rvert \le \lvert P \wedge Q\rvert\}$	$t_Q \ge t_{P\wedge Q}$

[Sha92] also proposes a model for real-time systems using the *since* operator embedded in PVS' higher-ordered logic. The theorem proving techniques that are demonstrated in his work show that the *since* operator can be used to obtain real-time properties proofs. [MA99] show the properties of the *since* operator useful for axiomatic description needs which also is used for time dependent properties.

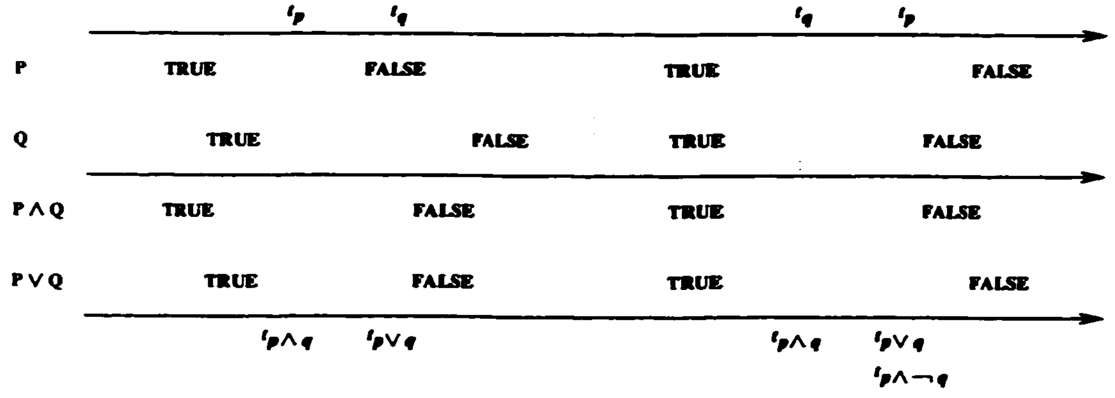

t_p t_q t_q t_p
P TRUE FALSE TRUE FALSE
Q TRUE FALSE TRUE FALSE
P∧Q TRUE FALSE TRUE FALSE
P∨Q TRUE FALSE TRUE FALSE
$t_{p\wedge q}$ $t_{p\vee q}$ $t_{p\wedge q}$ $t_{p\vee q}$
$t_{p\wedge\neg q}$

Figure 6: Absolute times at which predicates become false

3.5 TROM with *since* expressions

By instantiating the subset of the axioms of Section 2.4.1 in Chapter 2 with data from the formal specifications of a TROM design, we can derive a set of axioms based on the *since* expression to invlolve duration on state predicates and time intervals. Such is the proposed methodology given by [MA99].

3.5.1 Transition axioms

The objective of transition axioms is to state an ordering relation on the occurrence of two transitions in an object. By instantiating all transitions with the transition axiom from the logical semantics, we retain pairs of axiom instantiations where $S_2 = S_3$ and we obtain the following assuming that post-conditions hold for the transitions:

For all pairs where $S_2 = S_3$

$$A.HoldAt(S_1, t_1) \wedge A.Occur(e_1, p_i, t_1) \wedge (t_1 < t_2) \longrightarrow A.HoldAt(S_2, t_2)$$
$$A.HoldAt(S_3, t_3) \wedge A.Occur(e_1, p_i, t_3) \wedge (t_3 < t_4) \longrightarrow A.HoldAt(S_4, t_2)$$

Therefore the resulting *since* expression shows that after two transitions (ie: once the object is in the designation state (S_4) of the second transition), the time since the object was in state S_3 (or S_2 it is the same state) is smaller than the time since the object was in state S_1. We obtain the following *since* axiom:

$$A = S_4 \subset since(A = S_3) < since(A = S_1)$$

3.5.2 Time-constraint axioms

The objective of the time constraint axioms is to express the time interval during which a reaction to a transition is to occur. Each time constraint of a TROM has a lower and upper time limit that constrains the occurrence of a reaction within these limits after the firing transition occurrence. Again, instantiating the time constraint data of TROM objects with the appropriate logical semantic axiom, in this case the constrained event axiom, we obtain the following:

$A.Occur(e_1, p_i, t_a) \longrightarrow A.Occur(e_2, p_j, t_b) \wedge within(t_a, l, u, t_b)$
where t_a is the trigger event (e_1) occurrence time and t_b is the constrained event (e_2) occurrence time and where l and u are lower and upper bounds of the time constraint

Since we are describing the time relationship between two events, we have to describe the two transitions that are triggered by these two events. We use the transition axiom from the logical semantics to describe the two transitions as follow:

$$A.HoldAt(S_1, t_a) \wedge A.Occur(e_1, p_i, t_a) \wedge (t_a < t_b) \longrightarrow A.HoldAt(S_2, t_b)$$
$$A.HoldAt(S_3, t_c) \wedge A.Occur(e_2, p_i, t_c) \wedge (t_c < t_d) \longrightarrow A.HoldAt(S_4, t_d)$$

With these logical semantics axioms instantiated, we follow the following rules to extract the *since* expression.

A. if $S_2 = S_3$, that is if the constrained event is the next event after the firing transition:

$$A = S_4 \supset since(A = S_1) \; - \; since(A = S_2) \; < \; l$$
$$A = S_4 \supset since(A = S_1) \; - \; since(A = S_2) \; > \; u$$

B. if $S_2 \neq S_3$ that is if the constrained event does not follow immediately after the firing transition, then for each state *between* S_2 and S_3 we follow the following: Two cases to consider:

(a) $S_1 = S_4$

$$A = S_2 \supset since(A = S_1) \; < \; u$$
$$A = S_3 \supset since(A = S_1) \; < \; u$$

and for all states S between S_2 and S_3

$$A = S \supset since(A = S_1) < u$$

(b) $S_1 \neq S_4$

$$A = S_4 \supset since(A = S_1) \; - \; since(A = S_3) \; < \; u$$
$$A = S_4 \supset since(A = S_1) \; - \; since(A = S_3) \; > \; l$$

and for all states S between S_2 and S_3

$$A = S \supset since(A = S_1) \; < \; u$$

3.5.3 Synchronization axioms

The objective of synchrony axioms is to express simultaneous change of state within two communicating objects. For each pair of communicating objects and with each associated external events (incoming, outgoing) we instantiate the synchrony axiom with the TROM information which results in the following:

$$A.Occur(e_A, p_A, t) \Leftrightarrow B.Occur(e_B, p_B, t)$$

Since we are describing the time relationship between two events, we have to describe the two transitions that are triggered by these two events. We use the transition axiom from the logical semantics to describe the two transitions as follow:

$$A.HoldAt(S_1, t_1) \wedge A.Occur(e, p_A, t_1) \wedge (t_1 < t_2) \longrightarrow A.HoldAt(S_2, t_2)$$
$$B.HoldAt(S_3, t_3) \wedge B.Occur(e, p_B, t_3) \wedge (t_3 < t_4) \longrightarrow B.HoldAt(S_4, t_4)$$

The ensuing *since* expression shows that two communicating objects with the occurrence of the same event (one incoming, one outgoing) have their triggered transitions at the same time. Therefore when object A is in state S_2 and when object B is in state S_4, the time since object A left state S_1 is equal to the time since object B left state S_3. We get the following *since* expression:

$$A = S_2 \wedge B = S_4 \subset since(A = S_1) = since(B = S_3)$$

A note has to be added on this derivation algorithm. If $S_1 = S_2$ or if $S_3 = S_4$ such an axiom can not be derived. This would apply to events triggering reflexive transitions. For example, assume $S_1 = S_2$, the above logical semantics axioms would hold true but the derived *since* axiom would not be true due to the fact that $since(A = S_2)$ would be equal to 0 and $since(B = S_3)$ would be equal to some value x greater than 0. The *since* axiom would hold true only if the event triggered reflexive transitions in both of the associated objects, giving $since(A = S_1) = since(B = S_3) = 0$

Chapter 4

From TROM to axiomatic description

In Chapter 2, we presented and explained the components of the TROM specifications and in Chapter 3, the axiomatic expression of these formal specifications were detailed. We saw the *since* operator approach of the axiomatic description and we saw the PVS theories axiomatic description of TROM specifications. In this chapter, the requirements for a mechanized derivation of these axioms will be presented, the design and the implementation details of the axiomatic description generator for both of these approaches. First, a component of the TROMLAB environment will be presented in order to better comprehend how the tool interacts within the environment.

4.1 Description of the AST

The abstract syntax tree (AST) is the internal representation of a TROM specification. This specification was syntactically checked as the AST is constructed. The AST, as its name implies, is a tree of links of all components of TROM classes and subsystems with access methods to all of these components. Figure 7 shows the high level structure of the AST with the components used by the generator developed. To navigate through and access information within the AST its developers have created all access operations needed, a sample of such operations is given in table 2. With this table, we see that we can retrieve all TROM data.

The development of the interpreter to construct the AST was originally done in

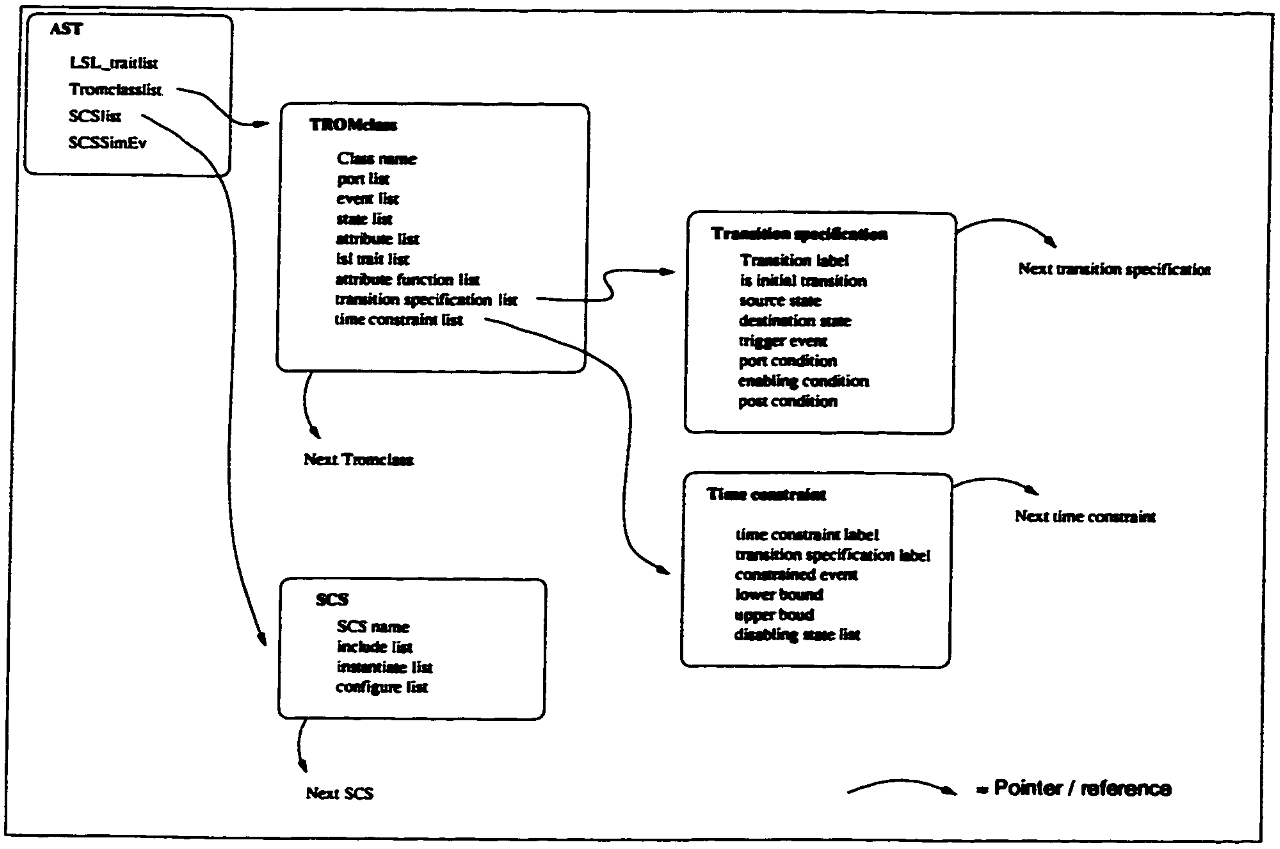


Figure 7: High level AST structure with subset of components shown

the C++ environment [Tao96] and then ported to Java [Sri99]. The axiom generator tool was done with the latest version.

4.2 From TROM to *since*

As we saw in Chapter 3, *since* is an interesting operator for duration measurement in the context of real-time behavior reasoning. We also saw the methodology that is involved to derive the *since* axiomatic description of TROMs. In this section the mechanization of the derivation process is explained.

4.2.1 Transition axioms

We saw in Section 3.5.1, we can derive *since* expressions from the TROM specifications. Figure 8 shows pseudo-code for the algorithm used in the axiom generation tool to extract and build the *since* transition axioms. The objective of the algorithm

Table 2: Sample AST operations

Access operation	*Informal signification*
AST.TROMclasslist.head()	returns the head of the TROM classes' list
(TROMclass).get_trans_speclist().head()	returns the head of the transition specification list of a TROM
(time_sconstraint).lower()	returns the lower bound of the time constraint
(trans_spec).get_source_state().get_state_name()	returns the name of the source state of the transition

is to find all consecutive transitions. In other words, isolate all pairs of transitions that come one after the other and generate the axiom for each of these pairs. Some restrictions must be applied in this algorithm which are to check:

- one of the transitions is reflexive
- it is not a repeated axioms due to parallel transitions. What is meant by parallel is two distinct transitions that involve the same source and destination state.

4.2.2 Time constraint axioms

The time-constraint axiom generation simple goes through all time constraints and creates the since inequality with both involved transitions specified in the time constraint. That is, the firing transition and the transition triggered by the constrained event. As stated in Chapter 3, the time-constraint axioms can include more inequalities depending on the relationship between the firing transition and the transition triggered by the constrained event. In Figure 10 the *between* algorithm is introduced. *between* is a function that returns all states on any paths of a state machine in between to states. If we take for example Figure 11, stating $between(A, E)$ would return the statelist C, E; $between(A, F)$ would return C, D, E, G and so on. In other words, it returns the list of states member of all possible paths between two states.First, here are the definitions of the main components of the between algorithm:

- $edge(A, B)$ is true if there exists a transition from A to B
- $next(A)$ is the set of states S such that $edge(A, S)$ is true
- $path(B, Z)$ is true if there exists a sequence of transitions from B to Z

```
while trom != null
      trans_spec1 = TROM.transition_specification_list.head()
      while trans_spec1 != null
            S1 = trans_spec1.source
            S2 = trans_spec1.destination
            trans_spec2 = TROM.transition_specification_list.head()
            while trans_spec2 != null
                  S3 = trans_spec2.source
                  S4 = trans_spec2.destination
                  if S2 == S3 AND
                    S1 != S4 AND
                    S1 != S3 AND
                    S2 != S4 AND
                    axiom has not yet been generated THEN
                           generate axiom :
                           "(Object = S4) implies since(Object = S3) < since(Object=S1)"
                  end if
            end while
      end while
end while
```

Figure 8: Pseudo-code for Transition specification *since* axioms algorithm for a TROM

```
trom = AST.tromlist.head()
while trom != null
      time_constraint = trom.time_constraintList.head()
      while time_constraint != null
            trans_spec = trom.trans_specList.head()
            if time_constraint.constr_event == trans_spec.trig_event
                call algorithm in Figure 10 with (trans_spec and transition stated in time_constraint)
            end if
      end while
end while
```

Figure 9: Pseudo-code for time constraint *since* axioms algorithm for a TROM

```
S1 = trans_spec.source()
S2 = trans_spec.destination()
S3 = time_constraint.transition.source()
S4 = time_constraint.transition.destination()
if S2 == S3
    output axiom with "Object = S4 -> since(object = S1) -
                                      since(object = S3) > lower bound of time constraint"
                      "Object = S4 -> since(object = S1) -
                                      since(object = S3) < upper bound of time constraint"
else if S1 == S4
    output axiom with "Object = S2 -> since(object = S1) < upper bound of time constraint"
    output axiom with "Object = S3 -> since(object = S1) < upper bound of time constraint"

    statelist = all states *between* S2 and S3
    *state* = statelist.head()
    while statelist != null
          output axiom with "Object = *state* -> since(object = state) <
                                                  upper bound of time constraint"
    end while
else
    output axiom with "Object = S4 -> since(object = S1) -
                      since(object = S3) > lower bound of time constraint"
                      "Object = S4 -> since(object = S1) -
                      since(object = S3) < upper bound of time constraint"
    statelist = all states *between* S2 and S3
    *state* = statelist.head()
    while statelist != null
          output axiom with "Object = *state* -> since(object = S1) <
                                                  upper bound of time constraint"
    end while
end if
```

Figure 10: Pseudo-code for time constraint secondary algorithm

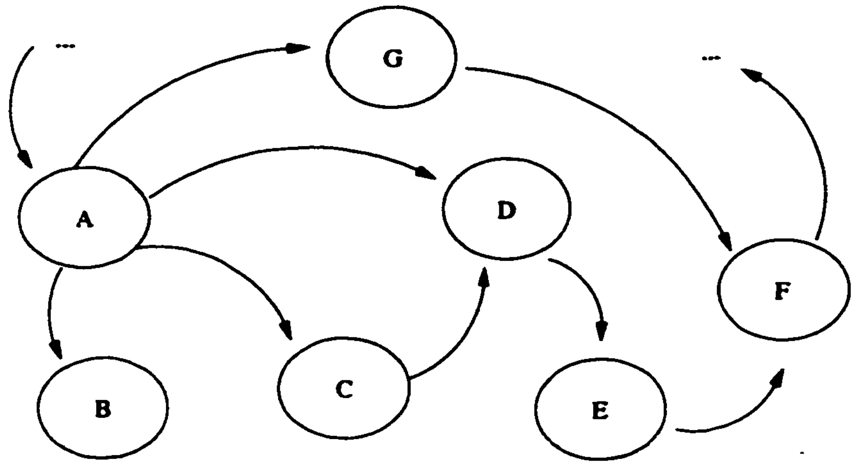


Figure 11: Statechart example

- *between*(A, Z) is the set of states S that are on all paths from A to Z

Here are the more formal definitions for the same components:

- $edge(A, B) = (\ \exists\ t\ \bullet\ t\ =$ non reflexive transition from A to B$)$
- $next(A)\ = \{\ S\ \mid\ edge(A, S)\ =\ true\ \}$
- $path(A, Z) = edge(A, Z) \vee (\ \exists\ S\ \bullet\ edge(A, S) \wedge path(S, Z)\ )$
- $between(A, Z) = \{\ S \mid S \in next(A) \wedge S \neq Z \wedge path(S, Z)\} \cup$
  $\{S \mid \exists S_2 \bullet S_2 \in next(A) \wedge S \in between(S_2, Z)\}$

Finally here is a formal definition of the two main algorithms $path(A, Z)$ and $between(A, Z)$:

```
$path(A, Z)$ :
        if $Z \in next(A)$ then return $true$
        elseif $A = Z$ then return $false$
        else
            $A.visited = true$
            return $(\exists\ S\ \bullet\ S\ \in next(A)\ \wedge\ S.visited = false\ \wedge\ path(S, Z) = true)$

$between(A, Z)$ :
          while $(\exists\ S\ \bullet\ S \in\ next(A) \wedge S.visited = false)$
                $S.visited = true$
            if ($S = Z$) then return { }
            elseif *path*($S, Z$) then
                return $\{S\} \cup$ *between*($S, Z$)
```

This algorithm is presented as an alternative to the first *path* algorithm. The latter was the selected alternative.

```
path(A, Z) : (non recursive)
    A.visited = true
    push(A)
    while stack not empty do
        while ($\exists S \mid S \in next(top()) \wedge S.visited = false$) do
            if ($S = Z$) then return true
            else
                S.visited = true
                push(S)
        pop()
    return false
```

4.2.3 Synchrony axioms

Synchrony axioms shown in Chapter 3 say that an outgoing event occurrence in one object corresponds to the incoming event occurrence within a communicating object. We also consider the fact that more than one event type can occur at a specific port type. Therefore the algorithm to extract from the AST the transitions to build the *since* axioms include a loop to repeat the axiom with other transitions that involve other events that are allowed through the same port-link configuration. The algorithm first builds a list of events allowed through the port, then then builds the axioms for each of those events.

At each of those events the following is applied: find the transitions T_A and T_B triggered by the said event, one in each object, say objects A and B. Use the source states and destination states of those transitions to build the axiom.

A = destination of T_A and B = destination of T_B implies that
since(A = source of T_A) = *since*(B = source of T_B)

An extra level in the algorithm is added when we add support for multiple transitions with the same triggering event within a single object. Hence, in the example of axiom stated above, we must consider adding a disjunction for different transitions but triggered by the same event. For example, if event e triggers one transition in object A and two transitions in object B, first find the transition in A, T_A, then find the transitions in B, $T1_B$ and $T2_B$. Then the following axiom is built:

$$\begin{aligned} &A = \text{destination of } T_A \text{ and } B = \text{destination of } T1_B \text{ or } T2_B \text{ implies that} \\ &\qquad \mathit{since}(A = \text{source of } T_A) = \mathit{since}(B = \text{source of } T1_B) \text{ or} \\ &\qquad \mathit{since}(A = \text{source of } T_A) = \mathit{since}(B = \text{source of } T2_B) \end{aligned}$$

Figure 12 shows pseudo code for the implemented algorithm for the synchrony *since* axiom generation. Keep in mind that this algorithm progressively builds the axiom as it traverses the AST. Unlike the two other axiom sets where output is done once, this progressively outputs the axioms.

4.2.4 *since* tool structure

In Figure 14 the class diagram of the implementation of the *since* axiom generator is depicted. Apendix A contains the definition of all the operations and attributes.

```
Assuming no reflexive transitions are involved.
configure = AST.SCS.ConfigureList.head()
while configure != null
   event == an event in the permitted events of the current port-link
   while event != null
     trans_spec1 = transition specifiation of the first object of the port-link
     while trans_spec1 != null
          if event == trigger event of trans_spec1
                 output : "object A = source of trans_spec1"
          trans_spec1 = trans_spec1.next
          end if
     end while
     output : "AND"
     trans_spec2 = transition specifiation of the second object of the port-link
     while trans_spec2 != null
          if event == trigger event of trans_spec2
                 output : "object B = source of trans_spec2"
          end if
          trans_spec2 = trans_spec2.next
     end while
     output : "implies"
     trans_spec1 = transition specifiation of the first object of the port-link
     while trans_spec1 != null
          if event == trigger event of trans_spec1
                 trans_spec2 = transition specifiation of the second object of the port-link
                 while trans_spec2 != null
                      if event == trigger event of trans_spec2
                             output : "since(object A = destination of trans_spec1) =
                                      since(object B = destination of trans_spec2)"
                      end if
                      trans_spec2 = trans_spec2.next
                 end while
       end if
                 trans_spec1 = trans_spec1.next
    end while
  end while
end while
```

Figure 12: Pseudo-code for Synchrony *since* axiom generation

4.3 From TROM to PVS

In Chapter 3, the axiomatic desription of the TROM specifications in the PVS environment has been described. We will now describe in a high level fashion the different processes that are applied to extract information from the AST. Each of these algorithms are used as working components of the application whose high level description will be given in the next section. We introduced a high-level descirption of the extracting algorithms in [AMP99] ealier. This section refines the algorithms in the context of a tool design.

The algorithm that extracts transition specifications and time constraint specifications from the TROM file(s) are to be applied for every object involved in the

```
Create AST
generate time constraint axioms
generate transition axioms
generate synchrony axioms
```

Figure 13: Pseudo-code for since main tool algorithm

model. As we saw in Chapter 3, a PVS theory is to be generated for every class in the model. Therefore, the algorithm to extract the transition specifications and the time constraint specifications will be applied for every TROM class.

4.3.1 Transition specifications

Transition axioms are to be extracted following a very straightforward algorithm. A transition specification is taken, say R_1, and its destination state is compared to all the transition specifications R_n (where $1 < n < number\ of\ transitions$) starting with the first transition specification. When the destination state of R_i matches the source state of R_n an axiom stating that the time of the triggering event of R_1 preceeds the time of the triggering event of R_n is to be generated. Figure 15 shows the pseudo code for this algorithm.

Transition axioms cannot be extracted freely and exhaustively. Users would have the remaining task of filtering through the generated axioms to make sure that conflicting axioms are not included. It is for this reason that some extra restrictions are to be applied while extracting and therefore commenting or removing all possibly conflicting axioms. What is meant by conflicting axiom is an axiom that is generated without regards to cycles within periods which could results in axioms contradicting other axioms. For example, a reflexive transition in the specifications would result in an axiom stating an inequality with the same event.

Here are the restriction rules when generating transition axioms:

1. **Axioms involving a single reflexive transition are excluded**: These cases are where the compared destination state and source state are from the same transition specification. The comparision between destination and source states matches but an axiom should not be generated since only one transition is involved. What is allowed though is if a transition specification is compared

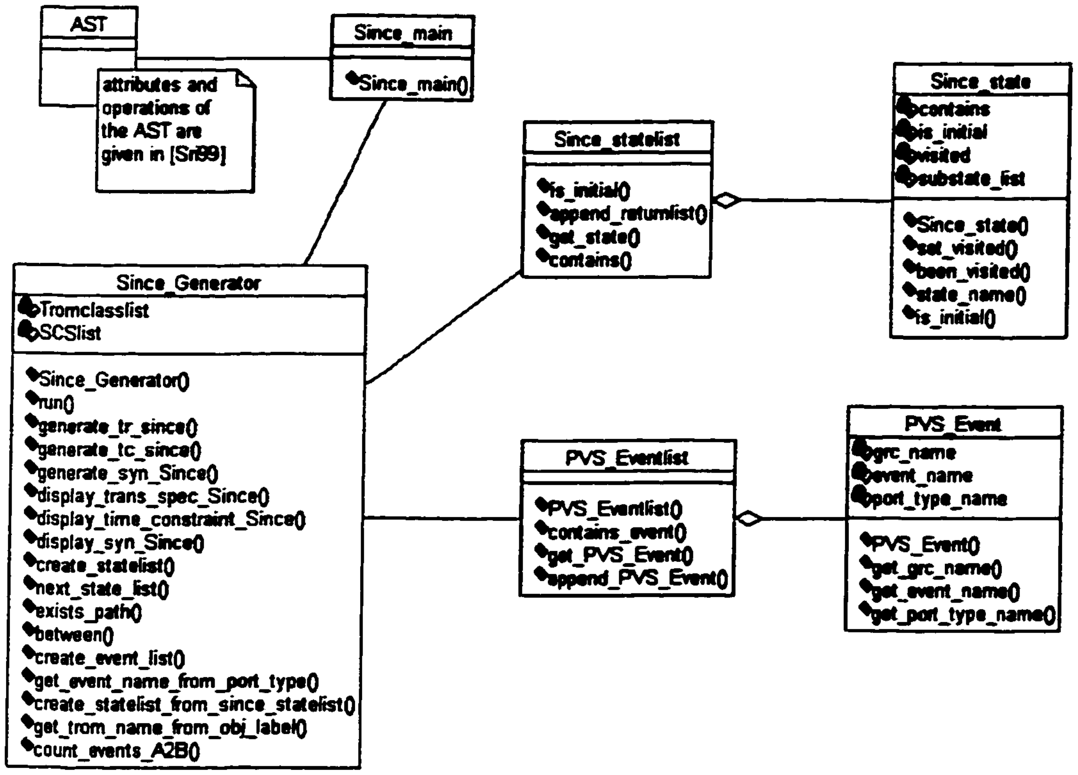


Figure 14: *since* axiom generator tool class diagram

with a reflexive transition. Since these are two distinct transition specifications, we can assume that one triggering event succeeds the other.

2. **Axioms involving the initial state as the destination state are excluded**: Since the model involves succeding periods of the state machine, we have to remove axioms that compare the "last" and "first" events of the periods. In other words, axioms involving transitions that lead to the initial state and transitions that leave the initial state must be excluded. Otherwise axioms stating that the last event of a period occurs before the first event of a period will be generated.

3. **Axioms involving the same event for two compared transitions are excluded**: If two succeeding transitions involve the same event, the generator will exclude the generation of such an axiom with such two transitions. Otherwise, an inequality based on the same event name would be generated, hence

this inequality could not hold.

4. **Axioms involving transitions with trigger events already stated in earlier axioms are commented out**: All axioms that are to be generated with events in the right hand side of the inequality already generated in an axiom where the same event is already generated in the left hand side of the inequality will be commented out. This leads to having different axiom lists depending on the ordering of the transition specification in the TROM formal specifications. This is why this restriction is not total. In other words, the axioms concerned with this restriction will be commented out so that the user can study the axiom set and make a decision that can not be incorporated in the algorithm of the tool.

```
trans_spec1 = TROM.transition_specification_list.head()
while trans_spec1 != null
      trans_spec2 = TROM.transition_specification_list.head()
      while trans_spec2 != null
            if trans_spec1.destination == trans_spec2.source AND
              trans_spec1 != trans_spec2 AND
              trans_spec1.trigger_event() != trans_spec2.trigger_event() AND
              trans_spec1.destination.initial_state() != true then
              if trans_spec2.trigger_event() is in temporary_event_list then comment out axiom
                generate axiom :
                     "TT(GRC)(i)(trans_spec1.trigger_event)(j) <
                                              TT(GRC)(i)(trans_spec2.trigger_event)(j)"
                add trans_spec1.trigger_event() to temporary_event_list
            end if
      end while
end while
```

Figure 15: Pseudo-code for Transition specification axioms algorithm for a TROM

4.3.2 Time constraint axioms

The algorithm to extract time constraint axioms generates one axiom per constrained reaction which are expressed in the time constraint section of the TROM class definition. Each time constraint is expressed as an event that must occur at a time t that is between a lower and upper bound stated in the time constraint. That time t is the difference between the time of the occrence of the constrained event and the time of the occurence of the event that triggered the transition stated in the time constraint.

Therefore the algorithm will generate an axiom per time constraint which will state that the difference between the occurence of the constrained event and the occurence of the event that triggers the transition stated in the same time constraint must be between the lower and upper bound also expressed in the same time constraint. Figure 16 shows the pseudo code for this algorithm.

```
time_constraint = TROM.time_constraint_list.head()
trans_spec = TROM.transition_specification_list.head()
while time_constraint != null
      transition_spec_name = time_constraint.get_name_of_transition_spec()
      while trans_spec != null OR found != true
      if transition_spec_name == name of trans_spec
            lower = time_constraint.lower()
            upper = time_constraint.upper()
            constr_event = time_constraint.constrained_event()
            trigg_event = trans_spec.trigger_event()
            generate axiom in PVS format:
                "TT(GRC)(i)(trigg_event)(j) - TT(GRC)(i)(constr_event)(j) > lower AND
                TT(GRC)(i)(trigg_event)(j) - TT(GRC)(i)(constr_event)(j) < upper"
            found = true
            trans_spec = transpec.next()
      end_while
      time_constraint = time_constraint.next()
end while
```

Figure 16: Pseudo-code for Time constraint axioms algorithm for a TROM

4.3.3 Synchrony axioms

The synchrony axioms are generated in a single PVS theory that corresponds to the subsytem of the model. Each axiom is a representation of the communication channels defined by the port-links in the configuration of the subsystem. Each link is a represenation of the synchronous message passing between the objects. Therefore, the algorithm to extract the syncrhony axiom will apply only the the subsystem(s) defined in the TROM model. The algorithm scans through the list of port-link configuration and generates one axiom per allowable event in the port-link per port port-link. That is, every port-link will generate at least one axiom and more if more than one event is allowed at the port of the objects involed. Figure 17 shows the pseudo code for this algorithm.

```
SCS = AST.SCSlist.head()
while SCS != null
    Configuration = AST.SCS.configurationlist.head()
    while Configuration != null
        Event = Head of list of all allowable events on the port of
        the objects of current configuration
        while Event != null
            GRC_id1 = Identification of left object of configuration
            GRC_id2 = Identification of right object of configuration
            generate axiom in PVS format :
                "TT(GRC_id1)(i)(Event)(j) = TT(GRC_id2)(i)(Event)(j)"
            Event = Event.next()
        end while
        Configuration = Configuration.next()
    end while
    SCS = SCS.next()
end while
```

Figure 17: Pseudo-code for Synchrony axioms algorithm

4.3.4 Main tool algorithm

Figure 18 shows the high level algorithm for the generation of the PVS theories.

```
Create AST Generate generic theories
trom = AST.tromlist.head()
while trom != null
      generate the GRC specific information for a theory
      generate time constraint axioms
      generate transition axioms
      trom = trom.next()
end while
generate synchrony axioms
```

Figure 18: Pseudo-code for main tool algorithm

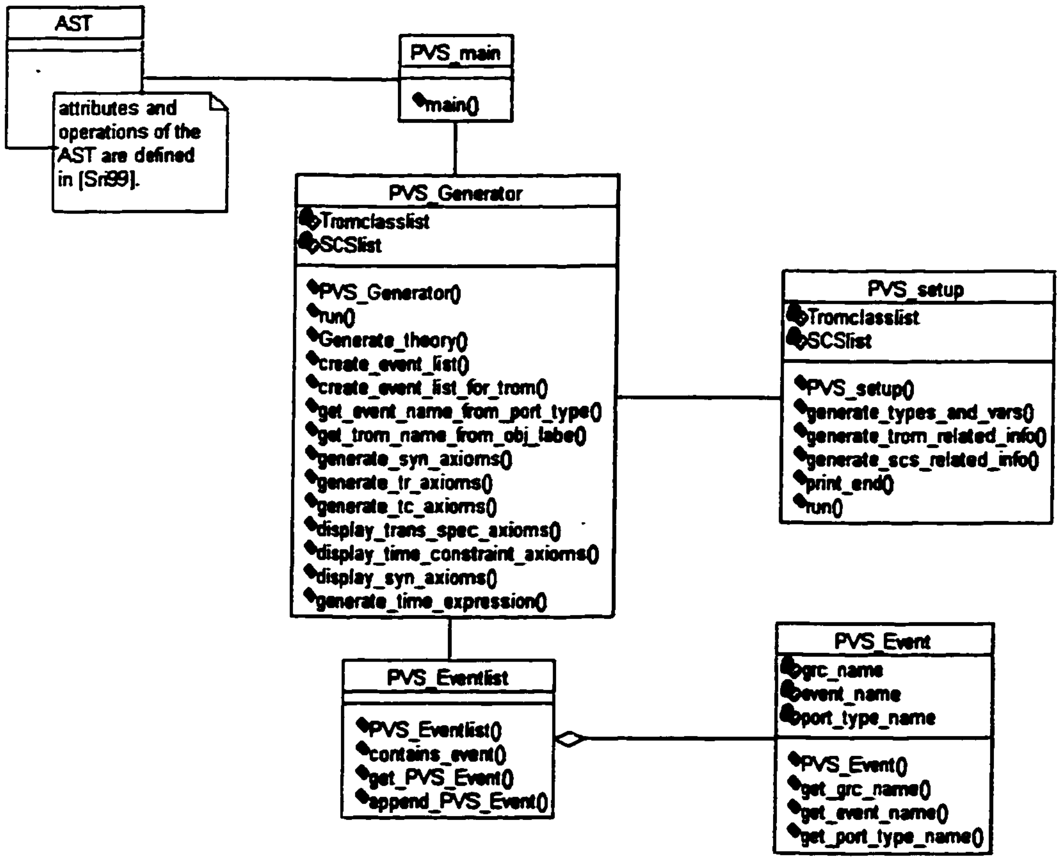


Figure 19: PVS generator tool class diagram

Chapter 5

Case study: Robotic Assembly System

This chapter will demonstrate the application of the axiomatic description generator. The model will be described informally and formally and then the application of the tool will be shown for both the PVS axiomatic description and the *since* axiomatic description.

In this Section we will illustrate the axiomatic description generator applied to a robotic assembly system. The problem will first be outlined informally followed by its formal counterpart in the TROM notation. Interleaved with these descriptions will be the transition, time constraint, synchrony and supplementary axioms as proposed by [AM99] and described in Chapter 3.

5.0.5 Problem description

The robotic assembly systems consists of six components all interacting together to obtain the assembly of parts that are submitted to it. The components are : a conveyor belt, a vision system, a right arm, a left arm, a stack and a tray. The belt provides the parts to the rest of the system where an assembly takes place (cup is placed over a dish) and then the assembled unit is deposited onto a tray. In order to allow for the safe pick-up of the parts, the belt stops for a pre-specified period of time whenever a part is in a pre-specified location known as pick-up zone. A sensor located under the belt detects the presence of a part. A part may be lost if it is not picked up in the pick-up zone. However, it is guaranteed that the parts on the belt are separated by a minimum distance in order to ensure a minimum time delay between two consecutive parts entering the pick-up zone. The parts may arrive on the belt in any order. It is required that for any arbitrary placement of n cups and n dishes on the belt, the system should produce n assemblies. This necessitates that no cup or dish placed on the belt be lost.

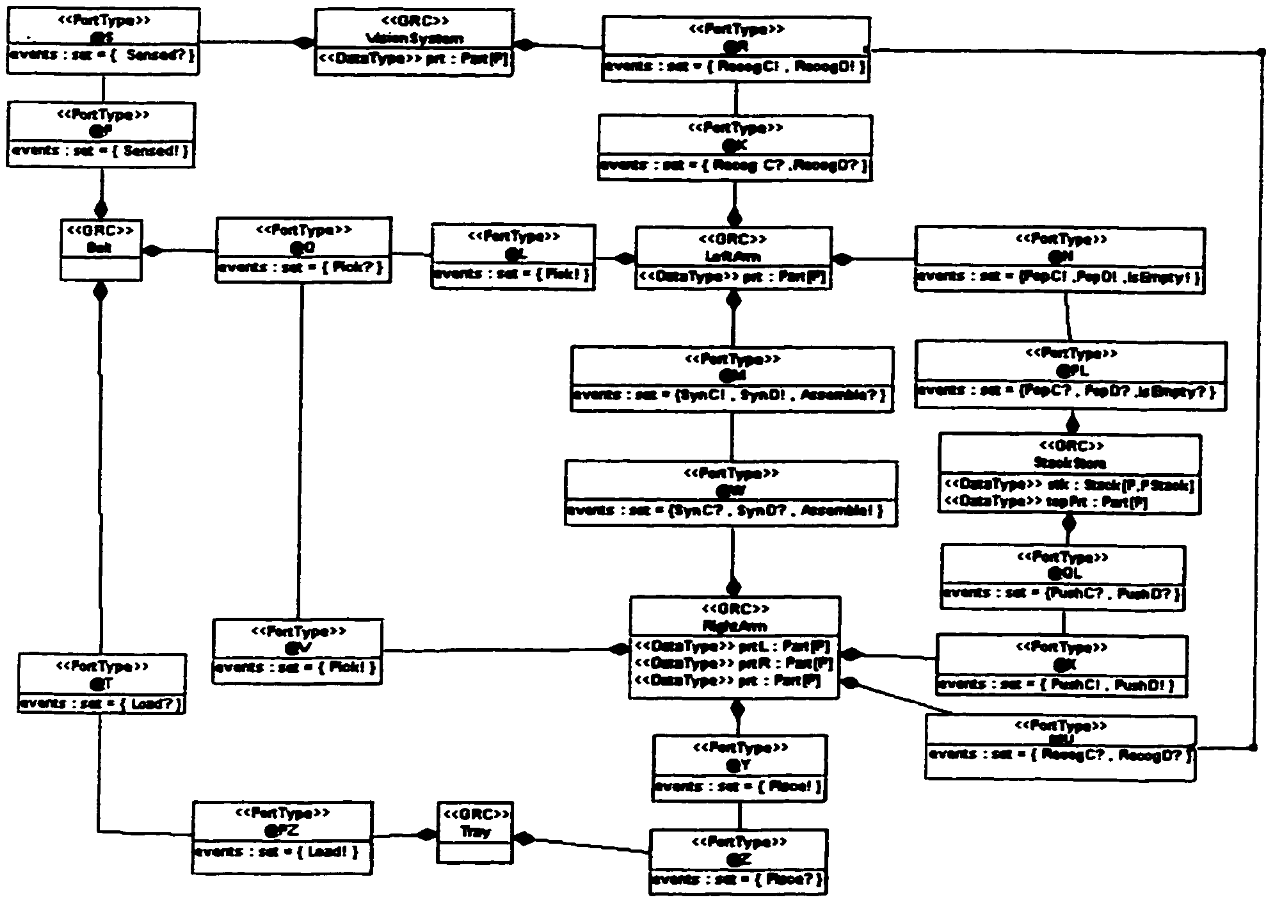
The system after the belt is composed of a vision system, left arm, a right arm and a stack. The vision system recognizes the incoming parts (cup or dish) and the stack stores the parts. Whenever a part comes into the view of the vision system's camera, scanning and recognition is performed by the vision system and it then signals the set of arms, within a maximal time delay constraint whether a cup or a dish has been recognized. This signal will activate the arms to perform the pick up of the part from the belt.

The arms use an algorithm based on a stack with which a part can be pushed or popped. Initially the left arm is free and the stack is empty. Whenever both arms are free and the stack is empty, if the vision systems signals the arms that a part is on the conveyor belt, the left arm picks up the part from the pick up zone. When the left arm holds a part, the right arm, if free, picks up the next part from the conveyor belt. If the part on the right arm, is the same as the part on the left arm, the part in the right arm is pushed into the stack. Otherwise, the parts are assembled and the resulting part is placed on a tray. If the left arm is free but the stack is not empty, the left arm picks up (pops) a part from the stack. In the design of the assembly process, the left arm is made free soon after the assembly while the right arm is made free only after placing the assembly on the tray.

For more details regarding the description of events, description of time constraints and description of the objects please refer to Chapter 2.

5.0.6 Robotic assembly system model

The behavior of the systems entities are modeled using generic reactive classes. Each of the GRC classes have a UML statechart diagram introduced in Figures 21, 23, 25, 27, 29 and 31. Their corresponding formal specifications are in Figures 22, 24, 26, 28, 30 and 32. Figure 20 shows all GRC classes with their corresponding PortType classes. Each association between the PortType classes shows the communication channels between the instances of the GRC classes. All formal specifications have been generated from UML by the Rose UML-TROM translator developed by [Pop99]. The collaboration diagram in Figure 33 shows the configuration of a robotic assembly system with one instance of each classes. Figure 34 shows its formal notation counterpart. This subsystem corresponds to a robotic assembly floor with one belt feeding the system and parts being output onto a single tray.



<<PortType>> @S events : set = { Sensed? }
<<GRC>> VisionSystem <<DataType>> prt : Part[P]
<<PortType>> @R events : set = { RecogC! , RecogD! }
<<PortType>> @P events : set = { Sensed! }
<<PortType>> @K events : set = { Recog C? ,RecogD? }
<<GRC>> Belt
<<PortType>> @O events : set = { Pick? }
<<PortType>> @L events : set = { Pick! }
<<GRC>> LeftArm <<DataType>> prt : Part[P]
<<PortType>> @N events : set = {PopC! ,PopD! ,IsEmpty! }
<<PortType>> @M events : set = {SynC! , SynD! , Assemble? }
<<PortType>> @PL events : set = {PopC? , PopD? ,IsEmpty? }
<<GRC>> StackStore <<DataType>> stk : Stack[P,PStack] <<DataType>> topPrt : Part[P]
<<PortType>> @W events : set = {SynC? , SynD? , Assemble! }
<<PortType>> @QL events : set = {PushC? , PushD? }
<<GRC>> RightArm <<DataType>> prtL : Part[P] <<DataType>> prtR : Part[P] <<DataType>> prt : Part[P]
<<PortType>> @V events : set = { Pick! }
<<PortType>> @X events : set = { PushC! , PushD! }
<<PortType>> @T events : set = { Load? }
<<PortType>> @U events : set = { RecogC? , RecogD? }
<<PortType>> @Y events : set = { Place! }
<<PortType>> @PZ events : set = { Load! }
<<GRC>> Tray
<<PortType>> @Z events : set = { Place? }

Figure 20: Robotic System class diagram

5.0.7 PVS axiomatic description

As [MA99] describe in their methodology, we begin with defining the set of events for each generic reactive classes.

```
Belt_Event : TYPE
  = {e_On, e_Sensed, e_Off, e_Load, e_Stop, e_Move, e_Pick}
VisionSystem_Event : TYPE
```

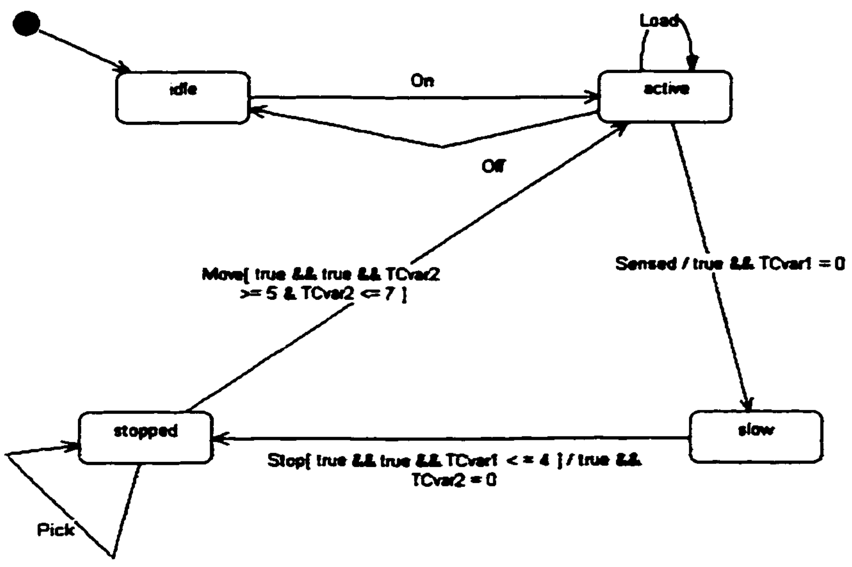

Load
idle
On
active
Off
Move[true && true && TCvar2 >= 5 & TCvar2 <= 7]
Sensed / true && TCvar1 = 0
stopped
slow
Stop[true && true && TCvar1 < = 4] / true && TCvar2 = 0
Pick

Figure 21: Belt state diagram

```
   = {e_Sensed, e_Unknown, e_Known, e_RecogD, e_RecogC}
StackStore_Event : TYPE
   = {e_PushC, e_PushD, e_PopC, e_PopD, e_IsEmpty}
LeftArm_Event : TYPE
   = {e_RecogD, e_RecogC, e_Pick, e_PopC, e_PopD,
                          e_IsEmpty, e_SynD, e_SynC, e_Free, e_Assemble}
RightArm_Event : TYPE
   = {e_Place, e_SynD, e_SynC, e_PushC, e_PushD,
                          e_Assemble, e_RecogD, e_RecogC, e_Pick}
Tray_Event : TYPE
   = {e_Load, e_Place}
```

For each class, a higher-order function is defined giving the transition time for an event occurrence, within a period, for an instance of a class. The signature of the functions are as follows once function *TT* has been overloaded with the theory *transition_time* containing the function definition.

```
TT: [Belt_GRC -> [Period -> [Belt_Event -> [Occurrence -> Time]]]]
TT: [VisionSystem_GRC -> [Period -> [VisionSystem_Event -> [Occurrence -> Time]]]]
TT: [StackStore_GRC -> [Period -> [StackStore_Event -> [Occurrence -> Time]]]]
TT: [LeftArm_GRC -> [Period -> [LeftArm_Event -> [Occurrence -> Time]]]]
TT: [RightArm_GRC -> [Period -> [RightArm_Event -> [Occurrence -> Time]]]]
TT: [Tray_GRC -> [Period -> [Tray_Event -> [Occurrence -> Time]]]]
```

```
Class Belt [@P, @Q, @T]
Events: On, Sensed!@P, Off, Load?@T, Stop, Move, Pick?@Q
States: *idle, active, slow, stopped
Attributes:
Traits:
Attribute-Function: idle → {};active → {};slow → {};stopped → {};
Transition-Specifications:
      R1: < idle, active >; On(true); true ⇒ true;
      R2: < active, slow >; Sensed(true); true ⇒ true;
      R3: < active, idle >; Off(true); true ⇒ true;
      R4: < active, active >; Load(true); true ⇒ true;
      R5: < slow, stopped >; Stop(true); true ⇒ true;
      R6: < stopped, active >; Move(true); true ⇒ true;
      R7: < stopped, stopped >; Pick(true); true ⇒ true;
Time-Constraints:
      TCvar1: R2, Stop, [0, 4], {};
      TCvar2: R5, Move, [5, 7], {};
end
```

Figure 22: Formal specification for Belt GRC

For the axioms in the transition axioms and time constraint axioms sections, i, j represent the i-th period in the computation of an object and the j-th occurrence of an event respectively.

Transition axioms

As the methodology described in Chapter 3, the transition axioms capture the ordering relation of the occurrences of events within a period of the object. Assuming that we can ignore the relationship of occurrences of events across periods we observe the following.

- In the Belt object, the events Sensed, Stop and Move occur only once per period but Pick and Load can have multiple occurrences within one period.
- In the Vision System object, all events can occur only once per period.
- In object StackStore, all events can have multiple occurrences.
- In object Left Arm, the events RecogC, RecogD, Pick and IsEmpty can occur only once per period but SynC, SynD, Assemble, Free, PopC and PopD can have multiple occurrences within one period.

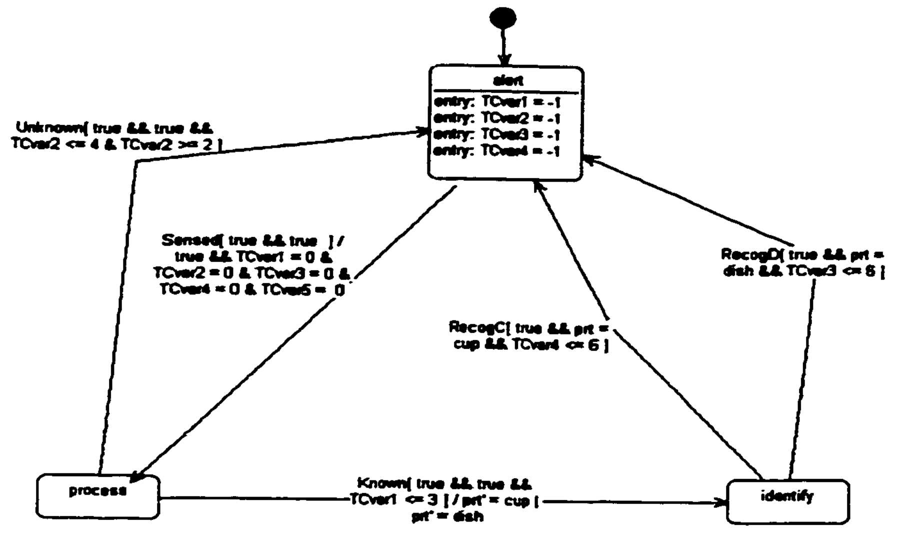

alert
entry: TCvar1 = -1
entry: TCvar2 = -1
entry: TCvar3 = -1
entry: TCvar4 = -1
Unknown[true && true && TCvar2 <= 4 & TCvar2 >= 2]
Sensed[true && true] / true && TCvar1 = 0 & TCvar2 = 0 & TCvar3 = 0 & TCvar4 = 0 & TCvar5 = 0
RecogC[true && prt = cup && TCvar4 <= 6]
RecogD[true && prt = dish && TCvar3 <= 6]
process
Known[true && true && TCvar1 <= 3] / prt' = cup | prt' = dish
identify

Figure 23: Vision System state diagram

- In object Right Arm, Assemble, Place, SynC and SynD can occur only once per period but RecogC, RecogD, Pick, PushC and PushD can occur more than once.

This information will be useful as we will discover that axiomatic description cannot be used to establish relationships of occurrence ordering when an event with multiple occurrences within a single period is involved in the relationship.

Belt class

1. The occurrence of event Sensed precedes the occurrence of Unknown within a period i, of an object Belt, Belt_VAR.

```
TR_AX_1 : AXIOM TT(Belt_VAR)(i)(e_Sensed)(1) < TT(Belt_VAR)(i)(e_Stop)(1)
```

2. The occurrence of event Stop precedes the occurrence of Move within a period i, of an object Belt, Belt_VAR.

```
TR_AX_2 : AXIOM TT(Belt_VAR)(i)(e_Stop)(1) < TT(Belt_VAR)(i)(e_Move)(1)
```

3. The occurrence of event Stop precedes all occurrences of Pick within a period i, of an object Belt, Belt_VAR.

```
TR_AX_3 : AXIOM TT(Belt_VAR)(i)(e_Stop)(1) < TT(Belt_VAR)(i)(e_Pick)(j)
```

4. All occurrences of event Pick precede the occurrence of Move within a period i, of an object Belt, Belt_VAR.

```
TR_AX_4 : AXIOM TT(Belt_VAR)(i)(e_Pick)(j) < TT(Belt_VAR)(i)(e_Move)(1)
```

Class VisionSystem [@R, @S]
Events: Sensed?@S, Unknown, Known, RecogD!@R, RecogC!@R
States: $^*alert, process, identify$
Attributes: prt:P
Traits: Part[P]
Attribute-Function: alert $\rightarrow$ {}; process $\rightarrow$ {}; identify $\rightarrow$ $\{prt\}$;
Transition-Specifications:
    R1: $< alert, process >$; Sensed(true); true $\Rightarrow$ true;
    R2: $< process, alert >$; Unknown(true); true $\Rightarrow$ true;
    R3: $< process, identify >$; Known(true); true $\Rightarrow$ prt'=cup | prt'=dish;
    R4: $< identify, alert >$; RecogD(true); prt=dish $\Rightarrow$ true;
    R5: $< identify, alert >$; RecogC(true); prt=cup $\Rightarrow$ true;
Time-Constraints:
    TCvar1: R1, Known, [0, 3], $\{alert\}$;
    TCvar2: R1, Unknown, [2, 4], $\{alert\}$;
    TCvar3: R1, RecogD, [0, 6], $\{alert\}$;
    TCvar4: R1, RecogC, [0, 6], $\{alert\}$;
end

Figure 24: Formal specification for Vision System GRC

Vision System class

1. The occurrence of event Sensed precedes the occurrences of Unknown within a period i, of an object Vision System, VisionSystem_VAR.

```
TR_AX_1 : AXIOM TT(VisionSystem_VAR)(i)(e_Sensed)(1) < TT(VisionSystem_VAR)(i)(e_Unknown)(1)
```

2. The occurrence of event Sensed precedes the occurrence of Known within a period i, of an object Vision System, VisionSystem_VAR.

```
TR_AX_2 : AXIOM TT(VisionSystem_VAR)(i)(e_Sensed)(1) < TT(VisionSystem_VAR)(i)(e_Known)(1)
```

3. The occurrence of event Known precedes the occurrence of RecogC within a period i, of an object Vision System, VisionSystem_VAR.

```
TR_AX_3 : AXIOM TT(VisionSystem_VAR)(i)(e_Known)(1) < TT(VisionSystem_VAR)(i)(e_RecogC)(1)
```

4. The occurrence of event Known precedes the occurrence of RecogD within a period i, of an object Vision System, VisionSystem_VAR.

```
TR_AX_3 : AXIOM TT(VisionSystem_VAR)(i)(e_Known)(1) < TT(VisionSystem_VAR)(i)(e_RecogD)(1)
```

StackStore class

- No transition axioms can be declared since all of the transitions go through the initial state, hence we cannot establish event occurrence ordering within a period.

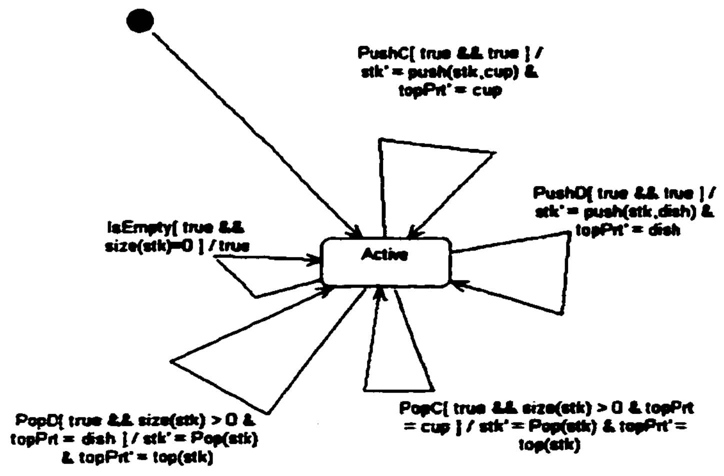


Figure 25: Stack state diagram

LeftArm class

1. The occurrence of event RecogC or RecogD precedes the occurrence of Pick within a period j, of an object Left Arm, LeftArm_VAR.

```
TR_AX_1 : AXIOM TT(LeftArm_VAR)(j)(e_RecogD)(1) < TT(LeftArm_VAR)(j)(e_Pick)(1)
TR_AX_2 : AXIOM TT(LeftArm_VAR)(j)(e_RecogC)(1) < TT(LeftArm_VAR)(j)(e_Pick)(1)
```

2. The occurrence of event Pick precedes all occurrences of SynC and SynD within a period j, of an object Left Arm, LeftArm_VAR.

```
TR_AX_3 : AXIOM TT(LeftArm_VAR)(j)(e_Pick)(1) < TT(LeftArm_VAR)(i)(e_SynD)(i)
TR_AX_4 : AXIOM TT(LeftArm_VAR)(j)(e_Pick)(1) < TT(LeftArm_VAR)(i)(e_SynC)(i)
```

3. Occurrence of event SynC, SynD precede occurrences of Assemble within a period j, of an object Left Arm, LeftArm_VAR. Occurrences are bound to the dishcup_occurrence supplementary axiom.

```
TR_AX_5 : AXIOM TT(LeftArm_VAR)(j)(e_SynD)(dish_occurrence) <
                              TT(LeftArm_VAR)(j)(e_Assemble)(dishcup_occurrence)
TR_AX_6 : AXIOM TT(LeftArm_VAR)(j)(e_SynC)(cup_occurrence) <
                              TT(LeftArm_VAR)(j)(e_Assemble)(dishcup_occurrence)
dishcup_occ_ax : AXIOM dishcup_occurrence =
dish_occurrence + cup_occurrence // see supplementary axioms
```

4. The i-th occurrence of event Assemble precedes the i-th occurrence of Free within a period j, of an object Left Arm, LeftArm_VAR.

```
TR_AX_7 : AXIOM TT(LeftArm_VAR)(j)(e_Assemble)(i) < TT(LeftArm_VAR)(j)(e_Free)(i)
```

```
Class StackStore [@PL, @QL]
Events: PushC?@QL, PushD?@QL, PopC?@PL, PopD?@PL, IsEmpty!@PL
States: *Active
Attributes: stk:PStack; topPrt:P
Traits: Stack[P,PStack],Part[P]
Attribute-Function: Active → {stk, topPrt};
Transition-Specifications:
    R1: < Active, Active >; PushC(true); true ⇒ stk'=push(stk,cup) & topPrt'=cup;
    R2: < Active, Active >; PushD(true); true ⇒ stk'=push(stk,dish) & topPrt'=dish;
    R3: < Active, Active >; PopC(true); size(stk)>0 & topPrt=cup ⇒ stk'=Pop(stk) & topPrt'=top(stk);
    R4: < Active, Active >; PopD(true); size(stk)>0 & topPrt=dish ⇒ stk'=Pop(stk) & topPrt'=top(stk);
    R5: < Active, Active >; IsEmpty(true); size(stk)=0 ⇒ true;
Time-Constraints:

end
```

Figure 26: Formal specification for StackStore GRC

5. The i-th occurrence of event Free precedes the i-th occurrence of PopC, PopD within a period j, of an object Left Arm, LeftArm_VAR. x, y are occurrences variables to capture occurrence relationship in cycles within a period.

```
TR_AX_8 : AXIOM TT(LeftArm_VAR)(j)(e_Free)(dishcup_occurrence) <
                              TT(LeftArm_VAR)(j)(e_PopC)(cup_occurrence)
TR_AX_9 : AXIOM TT(LeftArm_VAR)(j)(e_Free)(dishcup_occurrence) <
                              TT(LeftArm_VAR)(j)(e_PopD)(dish_occurrence)
dishcup_occ_ax : AXIOM dishcup_occurrence =
                              dish_occurrence + cup_occurrence // see supplementary axioms
```

6. Occurrences of event PopC, PopD precede occurrences of SynC, SynD within a period j, of an object Left Arm, LeftArm_VAR. The occurrences are bound to the popc_sync_ax and popd_synd_ax supplementary axioms

```
// TR_AX_10 : AXIOM TT(LeftArm_VAR)(j)(e_PopC)(pop_cup_occurrence)
                              < TT(LeftArm_VAR)(j)(e_SynD)(syn_dish_occurrence)
TR_AX_11 : AXIOM TT(LeftArm_VAR)(j)(e_PopC)(pop_cup_occurrence)
< TT(LeftArm_VAR)(j)(e_SynC)(syn_cup_occurrence)
TR_AX_12 : AXIOM TT(LeftArm_VAR)(j)(e_PopD)(pop_dish_occurrence)
                              < TT(LeftArm_VAR)(j)(e_SynD)(syn_dish_occurrence)
// TR_AX_13 : AXIOM TT(LeftArm_VAR)(j)(e_PopD)(pop_dish_occurrence)
                              < TT(LeftArm_VAR)(j)(e_SynC)(syn_cup_occurrence)
popc_sync_ax: AXIOM pop_cup_occurrence - syn_cup_occurrence < 2   // see supplementary axioms
popd_synd_ax: AXIOM pop_dish_occurrence - syn_dish_occurrence < 2 // see supplementary axiom
```

7. All occurrences of event Free precede the occurrence of IsEmpty within a period i, of an object Left Arm, LeftArm_VAR.

```
TR_AX_14 : AXIOM TT(LeftArm_VAR)(i)(e_Free)(j) < TT(LeftArm_VAR)(j)(e_IsEmpty)(1)
```

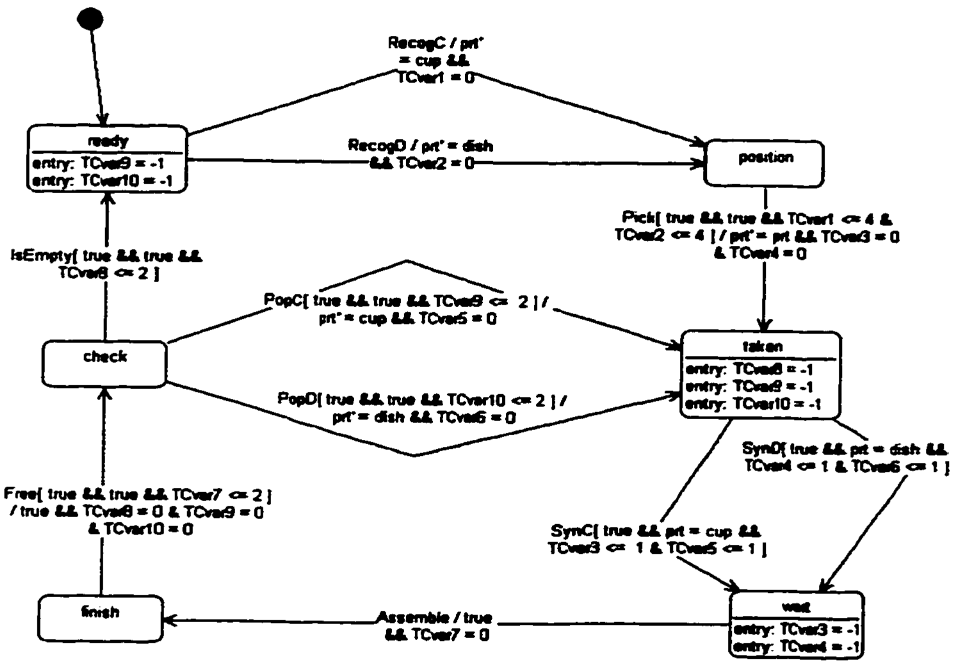

RecogC / prt' = cup && TCvar1 = 0
ready
entry: TCvar9 = -1
entry: TCvar10 = -1
RecogD / prt' = dish && TCvar2 = 0
position
Pick[true && true && TCvar1 <= 4 & TCvar2 <= 4] / prt' = prt && TCvar3 = 0 & TCvar4 = 0
IsEmpty[true && true && TCvar8 <= 2]
PopC[true && true && TCvar9 <= 2] / prt' = cup && TCvar5 = 0
check
taken
entry: TCvar8 = -1
entry: TCvar9 = -1
entry: TCvar10 = -1
PopD[true && true && TCvar10 <= 2] / prt' = dish && TCvar6 = 0
SynD[true && prt = dish && TCvar4 <= 1 & TCvar6 <= 1]
Free[true && true && TCvar7 <= 2] / true && TCvar8 = 0 & TCvar9 = 0 & TCvar10 = 0
SynC[true && prt = cup && TCvar3 <= 1 & TCvar5 <= 1]
finish
Assemble / true && TCvar7 = 0
wait
entry: TCvar3 = -1
entry: TCvar4 = -1

Figure 27: Left Arm state diagram

RightArm class

1. The occurrence of SynC, SynD precedes occurrences of RecogC, RecogD within a period i of an object Right Arm, RightArm_VAR.

```
TR_AX_1 : AXIOM TT(RightArm_VAR)(i)(e_SynD)(1) < TT(RightArm_VAR)(i)(e_RecogD)(i)
TR_AX_2 : AXIOM TT(RightArm_VAR)(i)(e_SynD)(1) < TT(RightArm_VAR)(i)(e_RecogC)(i)
TR_AX_3 : AXIOM TT(RightArm_VAR)(i)(e_SynC)(1) < TT(RightArm_VAR)(i)(e_RecogD)(i)
TR_AX_4 : AXIOM TT(RightArm_VAR)(i)(e_SynC)(1) < TT(RightArm_VAR)(i)(e_RecogC)(i)
```

2. Occurrences of event RecogC or RecogD precede all occurrences of Pick within a period i, of an object Right Arm, RightArm_VAR. The occurrences are bound to the dishcup_occurrence supplementary axiom.

```
TR_AX_1 : AXIOM TT(RightArm_VAR)(i)(e_RecogD)(cup_occurrence) <
                                TT(RightArm_VAR)(i)(e_Pick)(dishcup_occurrence)
TR_AX_2 : AXIOM TT(RightArm_VAR)(i)(e_RecogC)(dish_occurrence) <
                                TT(RightArm_VAR)(i)(e_Pick)(dishcup_occurrence)
dishcup_occ_ax : AXIOM dishcup_occurrence =
                                dish_occurrence + cup_occurrence // see supplementary axioms
```

3. All occurrences of event Pick precede the occurrence of Assemble within a period i, of an object Right Arm, RightArm_VAR.

```
TR_AX_3 : AXIOM TT(RightArm_VAR)(i)(e_Pick)(i) < TT(RightArm_VAR)(i)(e_Assemble)(1)
```

Class LeftArm [@L, @M, @N, @K]
Events: RecogD?@K, RecogC?@K, Pick!@L, PopC!@N, PopD!@N, IsEmpty!@N, SynD!@M, SynC!@M, Free, Assemble?@M
States: $^*ready, position, check, taken, finish, wait$
Attributes: prt:P
Traits: Part[P]
Attribute-Function: ready $\rightarrow$ {};position $\rightarrow$ $\{prt\}$;check $\rightarrow$ {};taken $\rightarrow$ $\{prt\}$;finish $\rightarrow$ {};wait $\rightarrow$ {};
Transition-Specifications:

- R1: $< ready, position >$; RecogD(true); true $\Rightarrow$ prt'=dish;
- R2: $< ready, position >$; RecogC(true); true $\Rightarrow$ prt'=cup;
- R3: $< position, taken >$; Pick(true); true $\Rightarrow$ prt'=prt;
- R4: $< check, taken >$; PopC(true); true $\Rightarrow$ prt'=cup;
- R5: $< check, taken >$; PopD(true); true $\Rightarrow$ prt'=dish;
- R6: $< check, ready >$; IsEmpty(true); true $\Rightarrow$ true;
- R7: $< taken, wait >$; SynD(true); prt=dish $\Rightarrow$ true;
- R8: $< taken, wait >$; SynC(true); prt=cup $\Rightarrow$ true;
- R9: $< finish, check >$; Free(true); true $\Rightarrow$ true;
- R10: $< wait, finish >$; Assemble(true); true $\Rightarrow$ true;

Time-Constraints:

- TCvar1: R2, Pick, [0, 4],{};
- TCvar2: R1, Pick, [0, 4], {};
- TCvar3: R3, SynC, [0, 1], $\{wait\}$;
- TCvar4: R3, SynD, [0, 1], $\{wait\}$;
- TCvar5: R4, SynC, [0, 1], {};
- TCvar6: R5, SynD, [0, 1], {};
- TCvar7: R10, Free, [0, 2], {};
- TCvar8: R9, IsEmpty, [0, 2], $\{taken\}$;
- TCvar9: R9, PopC, [0, 2], $\{ready, taken\}$;
- TCvar10: R9, PopD, [0, 2], $\{ready, taken\}$;

end

Figure 28: Formal specification for Left Arm GRC

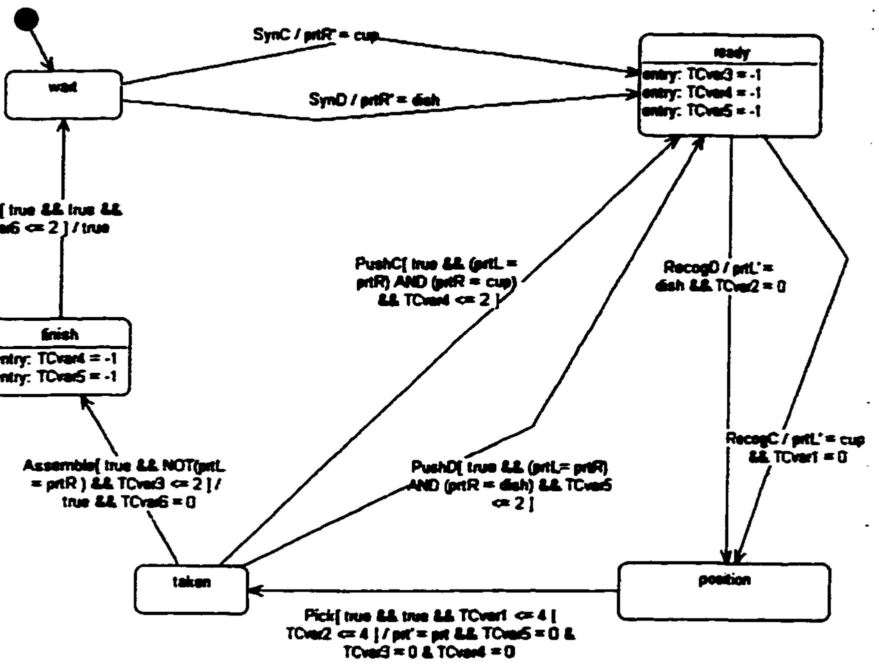

wait
ready
entry: TCvar3 = -1
entry: TCvar4 = -1
entry: TCvar5 = -1
SynC / prtR' = cup
SynD / prtR' = dish
Place[true && true && TCvar6 <= 2] / true
finish
entry: TCvar4 = -1
entry: TCvar5 = -1
PushC[true && (prtL = prtR) AND (prtR = cup) && TCvar4 <= 2]
RecogD / prtL' = dish && TCvar2 = 0
RecogC / prtL' = cup && TCvar1 = 0
Assemble[true && NOT(prtL = prtR) && TCvar3 <= 2] / true && TCvar6 = 0
PushD[true && (prtL= prtR) AND (prtR = dish) && TCvar5 <= 2]
taken
position
Pick[true && true && TCvar1 <= 4 | TCvar2 <= 4] / prt' = prt && TCvar5 = 0 & TCvar3 = 0 & TCvar4 = 0

Figure 29: Right Arm state diagram

4. Occurrences of event Pick precede occurrences of PushC, PushD within a period i, of an object Right Arm, RightArm_VAR. The occurrences are bound to the dishcup_occurrence supplementary axiom.

```
TR_AX_4 : AXIOM TT(RightArm_VAR)(i)(e_Pick)(dishcup_occurrence) <
                              TT(RightArm_VAR)(i)(e_PushC)(cup_occurrence)
TR_AX_5 : AXIOM TT(RightArm_VAR)(i)(e_Pick)(dishcup_occurrence) <
                              TT(RightArm_VAR)(i)(e_PushD)(dish_occurrence)
dishcup_occ_ax  : AXIOM dishcup_occurrence =
                              dish_occurrence + cup_occurrence // see supplementary axioms
```

5. Occurrences of PushC or PushD precede occurrences of RecogC or RecogD within a period i, of an object Right Arm, RightArm_VAR.The occurrences are bound to the push_recog_axiom supplementary axiom.

```
TR_AX_5 : AXIOM TT(RightArm_VAR)(i)(e_PushC)(push_cup_occ) <
                            TT(RightArm_VAR)(i)(e_RecogD)(Recog_dish_occ)
TR_AX_6 : AXIOM TT(RightArm_VAR)(i)(e_PushC)(push_cup_occ) <
                            TT(RightArm_VAR)(i)(e_RecogC)(Recog_cup_occ)
TR_AX_7 : AXIOM TT(RightArm_VAR)(i)(e_PushD)(push_dish_occ) <
                            TT(RightArm_VAR)(i)(e_RecogD)(Recog_dish_occ)
TR_AX_8 : AXIOM TT(RightArm_VAR)(i)(e_PushD)(push_dish_occ) <
                            TT(RightArm_VAR)(i)(e_RecogC)(Recog_cup_occ)
push_recog_axiom: AXIOM push_cup_occ + push_dish_occ =
                            Recog_cup_occ + Recog_dish_occ - 1 //see supplementary axioms
```

```
Class RightArm [@U, @Y, @X, @W, @V]
Events: Place!@Y, SynD?@W, SynC?@W, PushC!@X, PushD!@X, Assemble!@W, RecogD?@U,
RecogC?@U, Pick!@V
States: ready, finish,*wait, taken, position
Attributes: prtL:P; prtR:P; prt:P
Traits: Part[P]
Attribute-Function: finish → {};wait → {} ;taken→ {prt}; ready → {prtR}; position → {prtL};
Transition-Specifications:
      R1: < finish, wait >; Place(true); true ⇒ true;
      R2: < wait, ready >; SynD(true); true ⇒ prtR'=dish;
      R3: < wait, ready >; SynC(true); true ⇒ prtR'=cup;
      R4: < taken, ready >; PushC(true); (prtL=prtR) & (prtR=cup) ⇒ true;
      R5: < taken, ready >; PushD(true); (prtL=prtR) & (prtL=dish) ⇒ true;
      R6: < taken, finish >; Assemble(true); NOT(prtL=prtR) ⇒ true;
      R7: < ready, position >; RecogD(true); true ⇒ prtL'=dish;
      R8: < ready, position >; RecogC(true); true ⇒ prtL'=cup;
      R9: < position, taken >; Pick(true); true ⇒ prt'=prt;
Time-Constraints:
      TCvar1: R8, Pick, [0, 4], {};
      TCvar2: R7, Pick, [0, 4], {};
      TCvar3: R9, Assemble, [0, 2], {ready};
      TCvar4: R9, PushC, [0, 2], {finish, ready};
      TCvar5: R9, PushD, [0, 2], {finish, ready};
      TCvar6: R6, Place, [0, 2], {};
end
```

Figure 30: Formal specification for Right Arm GRC

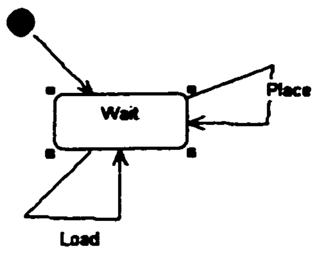


Figure 31: Tray state diagram

Class Tray [@Z, @PZ]
Events: Load!@PZ, Place?@Z
States: $*Wait, On$
Attributes:
Traits:
Attribute-Function: On → {}; Wait → {};
Transition-Specifications:
  R1: $< Wait, Wait >$; Place(true); true ⇒ true;
  R2: $< Wait, Wait >$; Load(true); true ⇒ true;
Time-Constraints:
end

Figure 32: Formal specification for Tray GRC

6. The occurrence of event Assemble precedes the occurrence of Place within a period i, of an object Right Arm, RightArm_VAR.

```
TR_AX_6 : AXIOM TT(RightArm_VAR)(i)(e_Assemble)(1) < TT(RightArm_VAR)(i)(e_Place)(1)
```

Tray class

- No transition axioms can be declared since all of the transitions go through the initial state, hence we cannot establish event occurrence ordering within a period.

Time constraint axioms

A time constraint axiom is included for each constrained reaction to a transition. An activated reaction, corresponding to the occurrence of an event, must occur within a specified time interval which is relative to the occurrence of the transition that has activated it. The following axioms capture all time constraints expressed in the specifications.

Belt class

1. The occurrence of event Stop in reaction to the occurrence of the event Sensed, occurs within an interval of 0 to 4 time units, within a period i and for the Belt object Belt_VAR.

```
TC_AX_1 : AXIOM TT(Belt_VAR)(i)(e_Stop)(j) - TT(Belt_VAR)(i)(e_Sensed)(j) > 0 AND
                TT(Belt_VAR)(i)(e_Stop)(j) - TT(Belt_VAR)(i)(e_Sensed)(j) < 4
```

2. The occurrence of event Move in reaction to the occurrence of the event Stop, occurs within an interval of 5 to 7 time units, within a period i and for the Belt object Belt_VAR.

```
TC_AX_2 : AXIOM TT(Belt_VAR)(i)(e_Move)(j) - TT(Belt_VAR)(i)(e_Stop)(j) > 5 AND
                TT(Belt_VAR)(i)(e_Move)(j) - TT(Belt_VAR)(i)(e_Stop)(j) < 7
```

VisionSystem class

1. The occurrence of event Known in reaction to the occurrence of the event Sensed, occurs within an interval of 0 to 3 time units, within a period i and for the VisionSystem object VisionSystem_VAR.

```
TC_AX_2 : AXIOM TT(VisionSystem_VAR)(i)(e_Known)(j) -
                          TT(VisionSystem_VAR)(i)(e_Sensed)(j) > 0 AND
                TT(VisionSystem_VAR)(i)(e_Known)(j) -
                          TT(VisionSystem_VAR)(i)(e_Sensed)(j) < 3
```

2. The occurrence of event Unknown in reaction to the occurrence of the event Sensed, occurs within an interval of 2 to 4 time units, within a period i and for the VisionSystem object VisionSystem_VAR.

```
TC_AX_1 : AXIOM TT(VisionSystem_VAR)(i)(e_Unknown)(j) -
                              TT(VisionSystem_VAR)(i)(e_Sensed)(j) > 2 AND
               TT(VisionSystem_VAR)(i)(e_Unknown)(j) -
                              TT(VisionSystem_VAR)(i)(e_Sensed)(j) < 4
```

3. The occurrence of event RecogC in reaction to the occurrence of the event Sensed, occurs within an interval of 0 to 6 time units, within a period i and for the VisionSystem object VisionSystem_VAR.

```
TC_AX_3 : AXIOM TT(VisionSystem_VAR)(i)(e_RecogD)(j) -
                              TT(VisionSystem_VAR)(i)(e_Sensed)(j) > 0 AND
               TT(VisionSystem_VAR)(i)(e_RecogD)(j) -
                              TT(VisionSystem_VAR)(i)(e_Sensed)(j) < 6
```

4. The occurrence of event RecogD in reaction to the occurrence of the event Sensed, occurs within an interval of 0 to 6 time units, within a period i and for the VisionSystem object VisionSystem_VAR.

```
TC_AX_4 : AXIOM TT(VisionSystem_VAR)(i)(e_RecogC)(j) -
                              TT(VisionSystem_VAR)(i)(e_Sensed)(j) > 0 AND
               TT(VisionSystem_VAR)(i)(e_RecogC)(j) -
                              TT(VisionSystem_VAR)(i)(e_Sensed)(j) < 6
```

StackStore class

- There are no time constraint expressed for the design of the class StackStore.

LeftArm class

1. The occurrence of event Pick in reaction to the occurrence of the event RecogC, occurs within an interval of 0 to 4 time units, within a period i and for the LeftArm object LeftArm_VAR.

```
TC_AX_1 : AXIOM TT(LeftArm_VAR)(i)(e_Pick)(j) -
                              TT(LeftArm_VAR)(i)(e_RecogC)(j) > 0 AND
               TT(LeftArm_VAR)(i)(e_Pick)(j) -
                              TT(LeftArm_VAR)(i)(e_RecogC)(j) < 4
```

2. The occurrence of event Pick in reaction to the occurrence of the event RecogD, occurs within an interval of 0 to 4 time units, within a period i and for the LeftArm object LeftArm_VAR.

```
TC_AX_2 : AXIOM TT(LeftArm_VAR)(i)(e_Pick)(j) -
                              TT(LeftArm_VAR)(i)(e_RecogD)(j) > 0 AND
               TT(LeftArm_VAR)(i)(e_Pick)(i) -
                              TT(LeftArm_VAR)(i)(e_RecogD)(j) < 4
```

3. The occurrence of event SynC in reaction to the occurrence of the event Pick, occurs within an interval of 0 to 1 time units, within a period i and for the LeftArm object LeftArm_VAR.

```
TC_AX_8 : AXIOM TT(LeftArm_VAR)(i)(e_SynC)(j) -
                              TT(LeftArm_VAR)(i)(e_Pick)(j) > 0 AND
                TT(LeftArm_VAR)(i)(e_SynC)(j) -
                              TT(LeftArm_VAR)(i)(e_Pick)(j) < 1
```

4. The occurrence of event SynD in reaction to the occurrence of the event Pick, occurs within an interval of 0 to 1 time units, within a period i and for the LeftArm object LeftArm_VAR.

```
TC_AX_6 : AXIOM TT(LeftArm_VAR)(i)(e_SynD)(j) -
                              TT(LeftArm_VAR)(i)(e_Pick)(j) > 0 AND
                TT(LeftArm_VAR)(i)(e_SynD)(j) -
                              TT(LeftArm_VAR)(i)(e_Pick)(j) < 1
```

5. The occurrence of event SynC in reaction to the occurrence of the event PopC, occurs within an interval of 0 to 1 time units, within a period i and for the LeftArm object LeftArm_VAR.

```
TC_AX_9 : AXIOM TT(LeftArm_VAR)(i)(e_SynC)(j) -
                              TT(LeftArm_VAR)(i)(e_PopC)(j) > 0 AND
                TT(LeftArm_VAR)(i)(e_SynC)(j) -
                              TT(LeftArm_VAR)(i)(e_PopC)(j) < 1
```

6. The occurrence of event SynD in reaction to the occurrence of the event PopD, occurs within an interval of 0 to 1 time units, within a period i and for the LeftArm object LeftArm_VAR.

```
TC_AX_7 : AXIOM TT(LeftArm_VAR)(i)(e_SynD)(j) -
                              TT(LeftArm_VAR)(i)(e_PopD)(j) > 0 AND
                TT(LeftArm_VAR)(i)(e_SynD)(j) -
                              TT(LeftArm_VAR)(i)(e_PopD)(j) < 1
```

7. The occurrence of event Free in reaction to the occurrence of the event Assemble, occurs within an interval of 0 to 2 time units, within a period i and for the LeftArm object LeftArm_VAR.

```
TC_AX_10 : AXIOM TT(LeftArm_VAR)(j)(e_Free)(i) -
                              TT(LeftArm_VAR)(i)(e_Assemble)(j) > 0 AND
                TT(LeftArm_VAR)(j)(e_Free)(i) -
                              TT(LeftArm_VAR)(i)(e_Assemble)(j) < 2
```

8. The occurrence of event IsEmpty in reaction to the occurrence of the event Free, occurs within an interval of 0 to 2 time units, within a period i and for the LeftArm object LeftArm_VAR.

```
TC_AX_5 : AXIOM TT(LeftArm_VAR)(i)(e_IsEmpty)(j) -
                            TT(LeftArm_VAR)(i)(e_Free)(j) > 0 AND
          TT(LeftArm_VAR)(i)(e_IsEmpty)(j) -
                            TT(LeftArm_VAR)(i)(e_Free)(j) < 2
```

9. The occurrence of event PopC in reaction to the occurrence of the event Free, occurs within an interval of 0 to 2 time units, within a period *i* and for the LeftArm object LeftArm_VAR.

```
TC_AX_3 : AXIOM TT(LeftArm_VAR)(i)(e_PopC)(j) -
                            TT(LeftArm_VAR)(i)(e_Free)(j) > 0 AND
          TT(LeftArm_VAR)(i)(e_PopC)(j) -
                            TT(LeftArm_VAR)(i)(e_Free)(j) < 2
```

10. The occurrence of event PopD in reaction to the occurrence of the event Free, occurs within an interval of 0 to 2 time units, within a period *i* and for the LeftArm object LeftArm_VAR.

```
TC_AX_4 : AXIOM TT(LeftArm_VAR)(i)(e_PopD)(j) -
                            TT(LeftArm_VAR)(i)(e_Free)(j) > 0 AND
          TT(LeftArm_VAR)(i)(e_PopD)(j) -
                            TT(LeftArm_VAR)(i)(e_Free)(j) < 2
```

RightArm class

1. The occurrence of event Pick in reaction to the occurrence of the event RecogC, occurs within an interval of 0 to 4 time units, within a period *i* and for the RightArm object RightArm_VAR.

```
TC_AX_5 : AXIOM TT(RightArm_VAR)(i)(e_Pick)(j) -
                            TT(RightArm_VAR)(i)(e_RecogC)(j) > 0 AND
          TT(RightArm_VAR)(i)(e_Pick)(j) -
                            TT(RightArm_VAR)(i)(e_RecogC)(j) < 4
```

2. The occurrence of event Pick in reaction to the occurrence of the event RecogD, occurs within an interval of 0 to 4 time units, within a period *i* and for the RightArm object RightArm_VAR.

```
TC_AX_6 : AXIOM TT(RightArm_VAR)(i)(e_Pick)(j) -
                            TT(RightArm_VAR)(i)(e_RecogD)(j) > 0 AND
          TT(RightArm_VAR)(i)(e_Pick)(j) -
                            TT(RightArm_VAR)(i)(e_RecogD)(j) < 4
```

3. The occurrence of event Assemble in reaction to the occurrence of the event Pick, occurs within an interval of 0 to 2 time units, within a period *i* and for the RightArm object RightArm_VAR.

```
TC_AX_4 : AXIOM TT(RightArm_VAR)(i)(e_Assemble)(j) -
                            TT(RightArm_VAR)(i)(e_Pick)(j) > 0 AND
          TT(RightArm_VAR)(i)(e_Assemble)(j) -
                            TT(RightArm_VAR)(i)(e_Pick)(j) < 2
```

4. The occurrence of event PushC in reaction to the occurrence of the event Pick, occurs within an interval of 0 to 2 time units, within a period i and for the RightArm object RightArm_VAR.

```
TC_AX_2 : AXIOM TT(RightArm_VAR)(i)(e_PushC)(j) -
                              TT(RightArm_VAR)(i)(e_Pick)(j) > 0 AND
            TT(RightArm_VAR)(i)(e_PushC)(j) -
                              TT(RightArm_VAR)(i)(e_Pick)(j) < 2
```

5. The occurrence of event PushD in reaction to the occurrence of the event Pick, occurs within an interval of 0 to 2 time units, within a period i and for the RightArm object RightArm_VAR.

```
TC_AX_3 : AXIOM TT(RightArm_VAR)(i)(e_PushD)(j) -
                              TT(RightArm_VAR)(i)(e_Pick)(j) > 0 AND
            TT(RightArm_VAR)(i)(e_PushD)(j) -
                              TT(RightArm_VAR)(i)(e_Pick)(j) < 2
```

6. The occurrence of event Place in reaction to the occurrence of the event Assemble, occurs within an interval of 0 to 2 time units, within a period i and for the RightArm object RightArm_VAR.

```
TC_AX_1 : AXIOM TT(RightArm_VAR)(i)(e_Place)(j) -
                              TT(RightArm_VAR)(i)(e_Assemble)(j) > 0 AND
            TT(RightArm_VAR)(i)(e_Place)(j) -
                              TT(RightArm_VAR)(i)(e_Assemble)(j) < 2
```

Tray class

- There are no time constraint expressed for the design of the class Tray.

Synchrony axioms

Message synchronization involves occurrences of input and output events that correspond in two object instances. In other words, the occurrence of an input event e in an instance of a class means that the same event e occurred as output event in another class instance. The following axioms are those that correspond to the configuration list of the subsystem described in Figure 34 and illustrated in the collaboration diagram in Figure 33.

1. Port $@S1$ of VS1 (class VisionSystem) is linked to port $@P1$ of object B1 (Belt class). All occurrences of event Sensed in VS1 occur simultaneously with Sensed in B1, in any period i.

```
SY_AX_1 : AXIOM TT(VS1)(i)(e_Sensed)(j) = TT(B1)(j)(e_Sensed)(j)
```

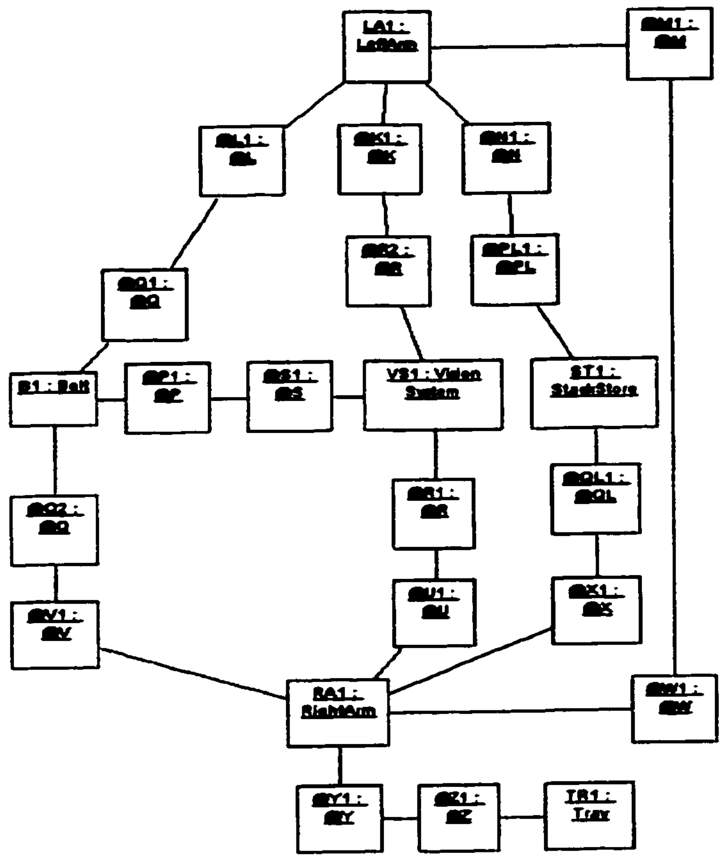


Figure 33: Collaboration diagram for robotic assembly system

2. Port $@V1$ of RA1 (class RightArm) is linked to port $@Q2$ of object B1 (class Belt). All occurrences of event Pick in RA1 occur simultaneously with Pick in B1, in any period i.

```
SY_AX_2 : AXIOM TT(RA1)(i)(e_Pick)(j) = TT(B1)(i)(e_Pick)(j)
```

3. Port $@PL1$ of ST1 (class StackStore) is linked to port $@N1$ of object LA1 (class LeftArm). All occurrences of event PopC in ST1 occur simultaneously with PopC in B1 and all occurrences of event PopC in ST1 occur simultaneously with PopC in B1 and all occurrences of event IsEmpty in ST1 occur simultaneously with IsEmpty in LA1, in any period i.

```
SY_AX_3 : AXIOM TT(ST1)(i)(e_PopC)(j) = TT(LA1)(i)(e_PopC)(j)
SY_AX_4 : AXIOM TT(ST1)(i)(e_PopD)(j) = TT(LA1)(i)(e_PopD)(j)
SY_AX_5 : AXIOM TT(ST1)(i)(e_IsEmpty)(j) = TT(LA1)(i)(e_IsEmpty)(j)
```

4. Port $@R2$ of VS1 (class VisionSystem) is linked to port $@K1$ of object LA1 (class LeftArm). All occurrences of event RecogD in VS1 occur simultaneously with RecogD in LA1 and all occurrences of event RecogC in VS1 occur simultaneously with RecogC in LA1, in any period i.

```
SCS robot
    Includes:
    Instantiate:
          B1::Belt[@P:1, @Q:2, @T:0];
          RA1::RightArm[@U:1, @Y:1, @X:1, @W:1, @V:1];
          ST1::StackStore[@PL:1, @QL:1];
          TR1::Tray[@Z:1, @PZ:1];
          LA1::LeftArm[@L:1, @M:1, @N:1, @K:1];
          VS1::VisionSystem[@R:2, @S:1];
    Configure:
          VS1.@S1:@S <=> B1.@P1:@P;
          RA1.@V1:@V <=> B1.@Q2:@Q;
          ST1.@PL1:@PL <=> LA1.@N1:@N;
          VS1.@R2:@R <=> LA1.@K1:@K;
          VS1.@R1:@R <=> RA1.@U1:@U;
          LA1.@M1:@M <=> RA1.@W1:@W;
          ST1.@QL1:@QL <=> RA1.@X1:@X;
          TR1.@Z1:@Z <=> RA1.@Y1:@Y;
          B1.@Q1:@Q <=> LA1.@L1:@L;
end
```

Figure 34: Subsystem for Robotic Assembly System

```
SY_AX_6 : AXIOM TT(VS1)(i)(e_RecogD)(j) = TT(LA1)(i)(e_RecogD)(j)
SY_AX_7 : AXIOM TT(VS1)(i)(e_RecogC)(j) = TT(LA1)(i)(e_RecogC)(j)
```

5. Port @$R1$ of VS1 (class VisionSystem) is linked to port @$U1$ of object RA1 (class RightArm). All occurrences of event RecogD in VS1 occur simultaneously with RecogD in RA1 and all occurrences of event RecogC in VS1 occur simultaneously with RecogC in RA1, in any period i.

```
SY_AX_8 : AXIOM TT(VS1)(i)(e_RecogD)(j) = TT(RA1)(i)(e_RecogD)(j)
SY_AX_9 : AXIOM TT(VS1)(i)(e_RecogC)(j) = TT(RA1)(i)(e_RecogC)(j)
```

6. Port @$M1$ of LA1 (class LeftArm) is linked to port @$W1$ of object RA1 (class RightArm). All occurrences of event SynD in LA1 occur simultaneously with SynD in RA1 and all occurrences of event SynC in LA1 occur simultaneously with SynC in RA1 and all occurrences of event Assemble in LA1 occur simultaneously with Assemble in RA1, in any period i.

```
SY_AX_10 : AXIOM TT(LA1)(i)(e_SynD)(j) = TT(RA1)(i)(e_SynD)(j)
SY_AX_11 : AXIOM TT(LA1)(i)(e_SynC)(j) = TT(RA1)(i)(e_SynC)(j)
SY_AX_12 : AXIOM TT(LA1)(i)(e_Assemble)(j) = TT(RA1)(i)(e_Assemble)(j)
```

7. Port @$QL1$ of ST1 (class StackStore) is linked to port @$X1$ of object RA1 (class RightArm). All occurrences of event PushC in ST1 occur simultaneously with PushC in RA1 and all occurrences of event PushD occur simultaneously with PushD in RA1, in a period i.

```
SY_AX_13 : AXIOM TT(ST1)(i)(e_PushC)(j) = TT(RA1)(i)(e_PushC)(j)
SY_AX_14 : AXIOM TT(ST1)(i)(e_PushD)(j) = TT(RA1)(i)(e_PushD)(j)
```

8. Port @$Z1$ of TR1 (class Tray) is linked to port @$Y1$ of object RA1 (class RightArm). All occurrences of event Place in TR1 occur simultaneously with Place in RA1, in a period i.

```
SY_AX_15 : AXIOM TT(TR1)(i)(e_Place)(j) = TT(RA1)(i)(e_Place)(j)
```

9. Port @$Q1$ of B1 (class Belt) is linked to port @$L1$ of object LA1 (class LeftArm). All occurrences of event Pick in B1 occur simultaneously with Pick in LA1, in a period i.

```
SY_AX_16 : AXIOM TT(B1)(i)(e_Pick)(j) = TT(LA1)(i)(e_Pick)(j)
```

5.0.8 Supplementary axioms

Supplementary axioms secure additional requirements or limit the reach of previously too general axioms. These are specific to the model at hand. In the case of the robotic assembly system, the fact that different parts can be picked up from the belt (cup or dish) is not precise enough in the previous axioms. Therefore we add the following axioms.

1. In many state transitions of the classes in the robotic assembly system, some transitions, say from a state S_i to S_j can be accomplished by two distinct transition specifications. Which are also complementary. That is, one or the other can occur due to fact that the robot handles a predetermined selection of parts, cup or dish. This supplementary axiom refines the fact that when two transitions are available to go from one state to the other, the sum of their occurrence history is equal to the occurrence history of the preceding transition if it is single as opposed to having two possible transitions. It can be used as occurrence variable in another time expression to establish the relationship between the event occurrence.

```
dishcup_occ_ax : AXIOM dishcup_occurrence = dish_occurrence + cup_occurrence
```

2. This axiom applies specifically to the LeftArm class. In order to compare the PopC, PopD triggered transitions with the SynC, SynD triggered transitions, the occurrence variable must again be directed by an axiom to establish proper relationship between transitions. We know that PopC will generate a SynC and PopD will generate a SynD. We know that SynD or SynC can have occurred zero time or once before PopD or PopC at the beginning of a period (depending on whether the part is taken from the stack or from the belt). We also know that PopC is followed by SynC and PopD by SynD. When PopC occurs for the first time, SynC will occur for the first or second time, the difference between their occurrence is zero or 1. This difference will apply for all subsequent occurrences within a period i, hence the popc_sync_ax axiom. The same rationale applies for the popd_synd_ax axiom.

```
popc_sync_ax: AXIOM pop_cup_occurrence - syn_cup_occurrence < 2
popd_synd_ax: AXIOM pop_dish_occurrence - syn_dish_occurrence < 2
```

3. Right Arm class. Again in order to establish relationship between two transitions that are doubled due to the cup/dish possibility, another supplementary axiom is given. When PushC or PushD occurs the next transition is not bound by a part selection, a new part is picked-up from the belt. Therefore the only relationship that we can establish between the occurrence numbers of PushC, PushD and RecogC, RecogD is that the sum of occurrences of PushC, PushD is equal to the sum of RecogC, RecogD minus 1. Minus 1 because RecogD or RecogC has occurred at least once before PushC or PushD after the initial state.

```
push_RecogC_axiom: AXIOM push_cup_occ + push_dish_occ = Recog_cup_occ + Recog_dish_occ - 1
```

5.1 Generated axiomatic description

This section includes an automatically generated PVS file that captures all statically perceptible information regarding transitions, time constraints and synchrony. The goal of this section is to comment the discrepancies between the automatically generated axioms and the manually generated axioms and obtain justification and establish the goals for further enhancements of the tool.

5.1.1 Generated PVS theories

```
Model: THEORY
    BEGIN
    Time  : TYPE = { r: real | r >= 0 }
    Episode : TYPE = posnat
    Occurrence : TYPE = posnat
END Model

transition_time[GRC: TYPE, GRC_Event: TYPE]: THEORY
    BEGIN
    IMPORTING Model
    TT: [GRC -> [Episode -> [GRC_Event -> [Occurrence -> Time]]]]
END transition_time

Tray: THEORY
    BEGIN
    Tray_GRC  : TYPE+
    Tray_Event  : TYPE = {e_Load, e_Place}
    IMPORTING transition_time[Tray_GRC,Tray_Event]
    i: VAR Episode
    j: VAR Occurrence
    Tray_VAR  : VAR Tray_GRC
END Tray

StackStore: THEORY
    BEGIN
    StackStore_GRC  : TYPE+
    StackStore_Event  : TYPE = {e_PushC, e_PushD, e_PopC, e_PopD, e_IsEmpty}
    IMPORTING transition_time[StackStore_GRC,StackStore_Event]
    i: VAR Episode
    j: VAR Occurrence
    StackStore_VAR  : VAR StackStore_GRC
END StackStore

RightArm: THEORY
    BEGIN
    RightArm_GRC  : TYPE+
    RightArm_Event  : TYPE = {e_Place, e_SynD, e_SynC, e_PushC, e_PushD, e_Assemble,
                              e_RecogD, e_RecogC, e_Pick}
    IMPORTING transition_time[RightArm_GRC,RightArm_Event]
    i: VAR Episode
    j: VAR Occurrence
    RightArm_VAR  : VAR RightArm_GRC
    TC_AX_1 : AXIOM TT(RightArm_VAR)(i)(e_Pick)(j) - TT(RightArm_VAR)(i)(e_RecogC)(j) > 0 AND
              TT(RightArm_VAR)(i)(e_Pick)(j) - TT(RightArm_VAR)(i)(e_RecogC)(j) < 4
    TC_AX_2 : AXIOM TT(RightArm_VAR)(i)(e_Pick)(j) - TT(RightArm_VAR)(i)(e_RecogD)(j) > 0 AND
              TT(RightArm_VAR)(i)(e_Pick)(j) - TT(RightArm_VAR)(i)(e_RecogD)(j) < 4
    TC_AX_3 : AXIOM TT(RightArm_VAR)(i)(e_Assemble)(j) - TT(RightArm_VAR)(i)(e_Pick)(j) > 0 AND
              TT(RightArm_VAR)(i)(e_Assemble)(j) - TT(RightArm_VAR)(i)(e_Pick)(j) < 2
    TC_AX_4 : AXIOM TT(RightArm_VAR)(i)(e_PushC)(j) - TT(RightArm_VAR)(i)(e_Pick)(j) > 0 AND
              TT(RightArm_VAR)(i)(e_PushC)(j) - TT(RightArm_VAR)(i)(e_Pick)(j) < 2
    TC_AX_5 : AXIOM TT(RightArm_VAR)(i)(e_PushD)(j) - TT(RightArm_VAR)(i)(e_Pick)(j) > 0 AND
              TT(RightArm_VAR)(i)(e_PushD)(j) - TT(RightArm_VAR)(i)(e_Pick)(j) < 2
    TC_AX_6 : AXIOM TT(RightArm_VAR)(i)(e_Place)(j) - TT(RightArm_VAR)(i)(e_Assemble)(j) > 0 AND
              TT(RightArm_VAR)(i)(e_Place)(j) - TT(RightArm_VAR)(i)(e_Assemble)(j) < 2
    TR_AX_1 : AXIOM TT(RightArm_VAR)(i)(e_SynD)(1) < TT(RightArm_VAR)(i)(e_RecogD)(1)
    TR_AX_2 : AXIOM TT(RightArm_VAR)(i)(e_SynD)(1) < TT(RightArm_VAR)(i)(e_RecogC)(1)
    TR_AX_3 : AXIOM TT(RightArm_VAR)(i)(e_SynC)(1) < TT(RightArm_VAR)(i)(e_RecogD)(1)
    TR_AX_4 : AXIOM TT(RightArm_VAR)(i)(e_SynC)(1) < TT(RightArm_VAR)(i)(e_RecogC)(1)
    TR_AX_5 : AXIOM TT(RightArm_VAR)(i)(e_PushC)(1) < TT(RightArm_VAR)(i)(e_RecogD)(1)
    TR_AX_6 : AXIOM TT(RightArm_VAR)(i)(e_PushC)(1) < TT(RightArm_VAR)(i)(e_RecogC)(1)
    TR_AX_7 : AXIOM TT(RightArm_VAR)(i)(e_PushD)(1) < TT(RightArm_VAR)(i)(e_RecogD)(1)
    TR_AX_8 : AXIOM TT(RightArm_VAR)(i)(e_PushD)(1) < TT(RightArm_VAR)(i)(e_RecogC)(1)
    TR_AX_9 : AXIOM TT(RightArm_VAR)(i)(e_Assemble)(1) < TT(RightArm_VAR)(i)(e_Place)(1)
    TR_AX_10 : AXIOM TT(RightArm_VAR)(i)(e_RecogD)(1) < TT(RightArm_VAR)(i)(e_Pick)(1)
    TR_AX_11 : AXIOM TT(RightArm_VAR)(i)(e_RecogC)(1) < TT(RightArm_VAR)(i)(e_Pick)(1)
%        TR_AX_12 : AXIOM TT(RightArm_VAR)(i)(e_Pick)(1) < TT(RightArm_VAR)(i)(e_PushC)(1)
%        TR_AX_13 : AXIOM TT(RightArm_VAR)(i)(e_Pick)(1) < TT(RightArm_VAR)(i)(e_PushD)(1)
%        TR_AX_14 : AXIOM TT(RightArm_VAR)(i)(e_Pick)(1) < TT(RightArm_VAR)(i)(e_Assemble)(1)
END RightArm

VisionSystem: THEORY
    BEGIN
    VisionSystem_GRC : TYPE+
    VisionSystem_Event : TYPE = {e_Sensed, e_Unknown, e_Known, e_RecogD, e_RecogC}
    IMPORTING transition_time[VisionSystem_GRC,VisionSystem_Event]
    i: VAR Episode
    j: VAR Occurrence
    VisionSystem_VAR : VAR VisionSystem_GRC
    TC_AX_1 : AXIOM TT(VisionSystem_VAR)(i)(e_Known)(j) - TT(VisionSystem_VAR)(i)(e_Sensed)(j) > 0 AND
              TT(VisionSystem_VAR)(i)(e_Known)(j) - TT(VisionSystem_VAR)(i)(e_Sensed)(j) < 3
    TC_AX_2 : AXIOM TT(VisionSystem_VAR)(i)(e_Unknown)(j) - TT(VisionSystem_VAR)(i)(e_Sensed)(j) > 2 AND
              TT(VisionSystem_VAR)(i)(e_Unknown)(j) - TT(VisionSystem_VAR)(i)(e_Sensed)(j) < 4
    TC_AX_3 : AXIOM TT(VisionSystem_VAR)(i)(e_RecogD)(j) - TT(VisionSystem_VAR)(i)(e_Sensed)(j) > 0 AND
              TT(VisionSystem_VAR)(i)(e_RecogD)(j) - TT(VisionSystem_VAR)(i)(e_Sensed)(j) < 6
    TC_AX_4 : AXIOM TT(VisionSystem_VAR)(i)(e_RecogC)(j) - TT(VisionSystem_VAR)(i)(e_Sensed)(j) > 0 AND
              TT(VisionSystem_VAR)(i)(e_RecogC)(j) - TT(VisionSystem_VAR)(i)(e_Sensed)(j) < 6
    TR_AX_1 : AXIOM TT(VisionSystem_VAR)(i)(e_Sensed)(1) < TT(VisionSystem_VAR)(i)(e_Unknown)(1)
    TR_AX_2 : AXIOM TT(VisionSystem_VAR)(i)(e_Sensed)(1) < TT(VisionSystem_VAR)(i)(e_Known)(1)
    TR_AX_3 : AXIOM TT(VisionSystem_VAR)(i)(e_Known)(1) < TT(VisionSystem_VAR)(i)(e_RecogD)(1)
    TR_AX_4 : AXIOM TT(VisionSystem_VAR)(i)(e_Known)(1) < TT(VisionSystem_VAR)(i)(e_RecogC)(1)
END VisionSystem
```

```
Belt: THEORY
  BEGIN
  Belt_GRC : TYPE+
  Belt_Event : TYPE = {e_Sensed, e_Load, e_Stop, e_Move, e_Pick}
  IMPORTING transition_time[Belt_GRC,Belt_Event]
  i: VAR Episode
  j: VAR Occurrence
  Belt_VAR : VAR Belt_GRC
  TC_AX_1 : AXIOM TT(Belt_VAR)(i)(e_Stop)(j) - TT(Belt_VAR)(i)(e_Sensed)(j) > 0 AND
            TT(Belt_VAR)(i)(e_Stop)(j) - TT(Belt_VAR)(i)(e_Sensed)(j) < 4
  TC_AX_2 : AXIOM TT(Belt_VAR)(i)(e_Move)(j) - TT(Belt_VAR)(i)(e_Stop)(j) > 5 AND
            TT(Belt_VAR)(i)(e_Move)(j) - TT(Belt_VAR)(i)(e_Stop)(j) < 7
  TR_AX_1 : AXIOM TT(Belt_VAR)(i)(e_Sensed)(1) < TT(Belt_VAR)(i)(e_Stop)(1)
  TR_AX_2 : AXIOM TT(Belt_VAR)(i)(e_Stop)(1) < TT(Belt_VAR)(i)(e_Move)(1)
  TR_AX_3 : AXIOM TT(Belt_VAR)(i)(e_Stop)(1) < TT(Belt_VAR)(i)(e_Pick)(1)
  TR_AX_4 : AXIOM TT(Belt_VAR)(i)(e_Pick)(1) < TT(Belt_VAR)(i)(e_Move)(1)
END Belt

 LeftArm: THEORY
  BEGIN
  LeftArm_GRC : TYPE+
  LeftArm_Event : TYPE = {e_RecogD, e_RecogC, e_Pick, e_PopC, e_PopD, e_IsEmpty, e_SynD,
                          e_SynC, e_Free, e_Assemble}
  IMPORTING transition_time[LeftArm_GRC,LeftArm_Event]
  i: VAR Episode
  j: VAR Occurrence
  LeftArm_VAR : VAR LeftArm_GRC
  TC_AX_1 : AXIOM TT(LeftArm_VAR)(i)(e_Pick)(j) - TT(LeftArm_VAR)(i)(e_RecogC)(j) > 0 AND
            TT(LeftArm_VAR)(i)(e_Pick)(j) - TT(LeftArm_VAR)(i)(e_RecogC)(j) < 4
  TC_AX_2 : AXIOM TT(LeftArm_VAR)(i)(e_Pick)(j) - TT(LeftArm_VAR)(i)(e_RecogD)(j) > 0 AND
            TT(LeftArm_VAR)(i)(e_Pick)(j) - TT(LeftArm_VAR)(i)(e_RecogD)(j) < 4
  TC_AX_3 : AXIOM TT(LeftArm_VAR)(i)(e_SynC)(j) - TT(LeftArm_VAR)(i)(e_Pick)(j) > 0 AND
            TT(LeftArm_VAR)(i)(e_SynC)(j) - TT(LeftArm_VAR)(i)(e_Pick)(j) < 1
  TC_AX_4 : AXIOM TT(LeftArm_VAR)(i)(e_SynD)(j) - TT(LeftArm_VAR)(i)(e_Pick)(j) > 0 AND
            TT(LeftArm_VAR)(i)(e_SynD)(j) - TT(LeftArm_VAR)(i)(e_Pick)(j) < 1
  TC_AX_5 : AXIOM TT(LeftArm_VAR)(i)(e_SynC)(j) - TT(LeftArm_VAR)(i)(e_PopC)(j) > 0 AND
            TT(LeftArm_VAR)(i)(e_SynC)(j) - TT(LeftArm_VAR)(i)(e_PopC)(j) < 1
  TC_AX_6 : AXIOM TT(LeftArm_VAR)(i)(e_SynD)(j) - TT(LeftArm_VAR)(i)(e_PopD)(j) > 0 AND
            TT(LeftArm_VAR)(i)(e_SynD)(j) - TT(LeftArm_VAR)(i)(e_PopD)(j) < 1
  TC_AX_7 : AXIOM TT(LeftArm_VAR)(i)(e_Free)(j) - TT(LeftArm_VAR)(i)(e_Assemble)(j) > 0 AND
            TT(LeftArm_VAR)(i)(e_Free)(j) - TT(LeftArm_VAR)(i)(e_Assemble)(j) < 2
  TC_AX_8 : AXIOM TT(LeftArm_VAR)(i)(e_IsEmpty)(j) - TT(LeftArm_VAR)(i)(e_Free)(j) > 0 AND
            TT(LeftArm_VAR)(i)(e_IsEmpty)(j) - TT(LeftArm_VAR)(i)(e_Free)(j) < 2
  TC_AX_9 : AXIOM TT(LeftArm_VAR)(i)(e_PopC)(j) - TT(LeftArm_VAR)(i)(e_Free)(j) > 0 AND
            TT(LeftArm_VAR)(i)(e_PopC)(j) - TT(LeftArm_VAR)(i)(e_Free)(j) < 2
  TC_AX_10 : AXIOM TT(LeftArm_VAR)(i)(e_PopD)(j) - TT(LeftArm_VAR)(i)(e_Free)(j) > 0 AND
            TT(LeftArm_VAR)(i)(e_PopD)(j) - TT(LeftArm_VAR)(i)(e_Free)(j) < 2
  TR_AX_1 : AXIOM TT(LeftArm_VAR)(i)(e_RecogD)(1) < TT(LeftArm_VAR)(i)(e_Pick)(1)
```

```
    TR_AX_2 : AXIOM TT(LeftArm_VAR)(i)(e_RecogC)(1)  < TT(LeftArm_VAR)(i)(e_Pick)(1)
    TR_AX_3 : AXIOM TT(LeftArm_VAR)(i)(e_Pick)(1)  < TT(LeftArm_VAR)(i)(e_SynD)(1)
    TR_AX_4 : AXIOM TT(LeftArm_VAR)(i)(e_Pick)(1)  < TT(LeftArm_VAR)(i)(e_SynC)(1)
    TR_AX_5 : AXIOM TT(LeftArm_VAR)(i)(e_PopC)(1)  < TT(LeftArm_VAR)(i)(e_SynD)(1)
    TR_AX_6 : AXIOM TT(LeftArm_VAR)(i)(e_PopC)(1)  < TT(LeftArm_VAR)(i)(e_SynC)(1)
    TR_AX_7 : AXIOM TT(LeftArm_VAR)(i)(e_PopD)(1)  < TT(LeftArm_VAR)(i)(e_SynD)(1)
    TR_AX_8 : AXIOM TT(LeftArm_VAR)(i)(e_PopD)(1)  < TT(LeftArm_VAR)(i)(e_SynC)(1)
    TR_AX_9 : AXIOM TT(LeftArm_VAR)(i)(e_SynD)(1)  < TT(LeftArm_VAR)(i)(e_Assemble)(1)
    TR_AX_10 : AXIOM TT(LeftArm_VAR)(i)(e_SynC)(1)  < TT(LeftArm_VAR)(i)(e_Assemble)(1)
%       TR_AX_11 : AXIOM TT(LeftArm_VAR)(i)(e_Free)(1)  < TT(LeftArm_VAR)(i)(e_PopC)(1)
%       TR_AX_12 : AXIOM TT(LeftArm_VAR)(i)(e_Free)(1)  < TT(LeftArm_VAR)(i)(e_PopD)(1)
    TR_AX_13 : AXIOM TT(LeftArm_VAR)(i)(e_Free)(1)  < TT(LeftArm_VAR)(i)(e_IsEmpty)(1)
%       TR_AX_14 : AXIOM TT(LeftArm_VAR)(i)(e_Assemble)(1)  < TT(LeftArm_VAR)(i)(e_Free)(1)
END LeftArm

   robot : THEORY
    BEGIN
    IMPORTING Tray, StackStore, RightArm, VisionSystem, Belt, LeftArm
    i: VAR Episode
    j: VAR Occurrence
    B1 : Belt_GRC
    RA1 : RightArm_GRC
    ST1 : StackStore_GRC
    TR1 : Tray_GRC
    LA1 : LeftArm_GRC
    VS1 : VisionSystem_GRC
    SY_AX_1 : AXIOM TT(VS1)(i)(e_Sensed)(j)  = TT(B1)(i)(e_Sensed)(j)
    SY_AX_2 : AXIOM TT(RA1)(i)(e_Pick)(j)  = TT(B1)(i)(e_Pick)(j)
    SY_AX_3 : AXIOM TT(ST1)(i)(e_PopC)(j)  = TT(LA1)(i)(e_PopC)(j)
    SY_AX_4 : AXIOM TT(ST1)(i)(e_PopD)(j)  = TT(LA1)(i)(e_PopD)(j)
    SY_AX_5 : AXIOM TT(ST1)(i)(e_IsEmpty)(j)  = TT(LA1)(i)(e_IsEmpty)(j)
    SY_AX_6 : AXIOM TT(VS1)(i)(e_RecogD)(j)  = TT(LA1)(i)(e_RecogD)(j)
    SY_AX_7 : AXIOM TT(VS1)(i)(e_RecogC)(j)  = TT(LA1)(i)(e_RecogC)(j)
    SY_AX_8 : AXIOM TT(VS1)(i)(e_RecogD)(j)  = TT(RA1)(i)(e_RecogD)(j)
    SY_AX_9 : AXIOM TT(VS1)(i)(e_RecogC)(j)  = TT(RA1)(i)(e_RecogC)(j)
    SY_AX_10 : AXIOM TT(LA1)(i)(e_SynD)(j)  = TT(RA1)(i)(e_SynD)(j)
    SY_AX_11 : AXIOM TT(LA1)(i)(e_SynC)(j)  = TT(RA1)(i)(e_SynC)(j)
    SY_AX_12 : AXIOM TT(LA1)(i)(e_Assemble)(j)  = TT(RA1)(i)(e_Assemble)(j)
    SY_AX_13 : AXIOM TT(ST1)(i)(e_PushC)(j)  = TT(RA1)(i)(e_PushC)(j)
    SY_AX_14 : AXIOM TT(ST1)(i)(e_PushD)(j)  = TT(RA1)(i)(e_PushD)(j)
    SY_AX_15 : AXIOM TT(TR1)(i)(e_Place)(j)  = TT(RA1)(i)(e_Place)(j)
    SY_AX_16 : AXIOM TT(B1)(i)(e_Pick)(j)  = TT(LA1)(i)(e_Pick)(j)
END robot2
```

5.1.2 Commenting PVS generated output

This section shows the differences between the manually derived axioms and the automatically derived axioms. Some axioms will be identified and a reason explaining the limitations of the tools will be given to justify the difference.

- All three supplementary axioms are not generated by the tool. As explained in the supplementary axiom section, 5.0.8 on page 64, these axioms are implicit to the nature of the problem. They are not explicitly described in the TROM specifications. Hence they are not automatically derived.
- Some axioms get to have their *occurrence* index controlled by the supplementary axioms. Hence the automatic derivation process cannot include these corrected indexes.
  - The transition axioms are generated with an index set to 1, since it is the safest assumption that can be made. That is, by expressing that the first occurrence of an event follows the first occurrence of the next one, we stay away from the pitfalls of the cycles problems.
  - The time-constraint axioms are generated with an index set to j. Again, because that would be the safest assumption. In the time-constraints' case, we assume that within a period each firing transitions correspond to a constrained event.
  - The synchrony axioms are also generated with an index set to j. We assume in these cases that all synchronized events have an equivalent occurrence in the other object.
- We see that some transition axioms are commented out, for example the ones on lines 70, 71 and 72. This is a voluntary mark made by the algorithm to alert designers that these axioms may be in conflict with previous axioms in the object theory. Hence, some revision by the designer is to be made. In this example, these three axioms are not in conflict and can therefore be uncommented. The same reasoning is to be done for all commented transition axioms.
- As stated above, the assumption for transition axioms *occurrence* representation is to leave set it to 1. We see in the axioms description of the transition

axiom section earlier, that axioms such as the ones on lines 59, 60, 61 and 62 can have the right part of the inequality set to j to show that many occurrences of RecogC or RecogD can occur after the first occurrence of SynD or SynC.

- All time constraint axioms are represented in the generated theories.
- All synchrony axioms are represented in the generated theories.

We see in this section that the main limitation of the tool is due to the supplementary axioms. Unfortunately, since this information is not found in the formal specifications of the TROM model, we cannot extract the information to generate the axioms.

Another limitation of the tool, is its inability to fully grasp cycles within state machines. That is, when a cycle within a computation period exists, the occurrence indexes must be revised by the designer of the specifications in order to ensure correct occurrence representation.

5.2 *since* axiomatic description

As we saw earlier, the *since* operator is also an appropriate tool to express time dependent properties of real time systems. In this section we will see the robotic assembly system expressed with the *since* operator. Every transition specification, time constraint and configuration specification in the subsystem can be expressed with the *since* operator. In other words, this section will again show a direct relationship between each a specification expression (transition, time constraint or synchrony) and a *since* expression. We will see in the next section that this direct relationship results in an automatable output from the formal specifications. This automated output is from the tool described in Chapter 4.

5.2.1 Transition axioms

Transition Axioms for the Right Arm

1. The RightArm enters the state position, received message RecogC or RecogD after pushing a cup or a dish or after receiving the SynC or SynD event. Applies for all subsequent states of *position* within a period.

```
RightArm(RightArm_GRC)(s0) = position v
    since(RightArm(RightArm_GRC)(s1)=ready) < since(RightArm(RightArm_GRC)(s2)=wait)
RightArm(RightArm_GRC)(s0) = position ->
    since(RightArm(RightArm_GRC)(s1)=ready) < since(RightArm(RightArm_GRC)(s2)=taken)
```

2. The RightArm enters the state *taken*, Picked a part from the belt, after receiving the RecogC or RecogD event.

```
RightArm(RightArm_GRC)(s0) = taken ->
    since(RightArm(RightArm_GRC)(s1)=position) < since(RightArm(RightArm_GRC)(s2)=ready)
```

3. The RightArm enters the state *finish*, Assembled a part, after picking up a part.

```
RightArm(RightArm_GRC)(s0) = finish ->
    since(RightArm(RightArm_GRC)(s1)=taken) < since(RightArm(RightArm_GRC)(s2)=position)
```

4. The RightArm enters the state *ready*, Pushed a cup or dish, after picking up a part or received *SynC* or *SynD* after sending *Place*. Counter example for subsequent.

```
RightArm(RightArm_GRC)(s0) = ready ->
    since(RightArm(RightArm_GRC)(s1)=wait) < since(RightArm(RightArm_GRC)(s2)=finish)
RightArm(RightArm_GRC)(s0) = ready ->
    since(RightArm(RightArm_GRC)(s1)=taken) < since(RightArm(RightArm_GRC)(s2)=position)
```

5. The RightArm enters the state *wait*, placed an assembled part on the tray, after assembling it.

```
RightArm(RightArm_GRC)(s0) = wait ->
    since(RightArm(RightArm_GRC)(s1)=finish) < since(RightArm(RightArm_GRC)(s2)=taken)
```

Transition Axioms for the LeftArm

1. The LeftArm enters the state *taken*, picked-up a part from the belt after receiving the RecogC or RecogD event.

```
LeftArm(LeftArm_GRC)(s0) = taken ->
    since(LeftArm(LeftArm_GRC)(s1)=position) < since(LeftArm(LeftArm_GRC)(s2)=ready)
```

2. The LeftArm enters the state *taken*, popped a cup or a dish after internal event *free*.

```
LeftArm(LeftArm_GRC)(s0) = taken ->
    since(LeftArm(LeftArm_GRC)(s1)=check) < since(LeftArm(LeftArm_GRC)(s2)=finish)
```

3. The LeftArm enters the state *wait*, sent the SynC or SynD event after popping a cup or a dish or after picking up a part from the belt.

```
LeftArm(LeftArm_GRC)(s0) = wait ->
    since(LeftArm(LeftArm_GRC)(s1)=taken) < since(LeftArm(LeftArm_GRC)(s2)=position)
LeftArm(LeftArm_GRC)(s0) = wait ->
    since(LeftArm(LeftArm_GRC)(s1)=taken) < since(LeftArm(LeftArm_GRC)(s2)=check)
```

4. The LeftArm enters the state *finish*, Assembled a part after sending the SynC or SynD event.

```
LeftArm(LeftArm_GRC)(s0) = finish ->
    since(LeftArm(LeftArm_GRC)(s1)=wait) < since(LeftArm(LeftArm_GRC)(s2)=taken)
```

5. The LeftArm enters the state *check*, internal event *free* after assembling the parts.

```
LeftArm(LeftArm_GRC)(s0) = check ->
    since(LeftArm(LeftArm_GRC)(s1)=finish) < since(LeftArm(LeftArm_GRC)(s2)=wait)
```

6. The LeftArm enters the state *ready*, sent the event *IsEmpty* after internal event *Free*.

```
LeftArm(LeftArm_GRC)(s0) = ready ->
    since(LeftArm(LeftArm_GRC)(s1)=check) < since(LeftArm(LeftArm_GRC)(s2)=finish)
```

7. The LeftArm enters the state *position*, received the RecogC or RecogD after sending the *IsEmpty*.

```
LeftArm(LeftArm_GRC)(s0) = position ->
    since(LeftArm(LeftArm_GRC)(s1)=ready) < since(LeftArm(LeftArm_GRC)(s2)=check)
```

Transition Axioms for the Belt

1. The Belt enters the state *active*, internal event *Move* after receiving the *Pick* event.

```
Belt(Belt_GRC)(s0) = active ->
    since(Belt(Belt_GRC)(s1)=stopped) < since(Belt(Belt_GRC)(s2)=slow)
```

2. The Belt enters the state *stopped*, received event *Stop* after sending event *Sensed*.

```
Belt(Belt_GRC)(s0) = stopped ->
    since(Belt(Belt_GRC)(s1)=slow) < since(Belt(Belt_GRC)(s2)=active)
```

3. The Belt enters the state *slow*, sent event *Sensed* after internal event *Move*

```
Belt(Belt_GRC)(s0) = slow ->
    since(Belt(Belt_GRC)(s1)=active) < since(Belt(Belt_GRC)(s2)=stopped)
```

Transition Axioms for the VisionSystem

1. The VisionSystem enters the state *identify*, internal event *Known* after incoming event *Sensed*.

```
VisionSystem(VisionSystem_GRC)(s0) = identify ->
    since(VisionSystem(VisionSystem_GRC)(s1)=process) <
                        since(VisionSystem(VisionSystem_GRC)(s2)=alert)
```

2. The VisionSystem enters the state *process*, incoming event *Sensed* after outgoing event *RecogC* or *RecogD*.

```
VisionSystem(VisionSystem_GRC)(s0) = alert ->
    since(VisionSystem(VisionSystem_GRC)(s1)=identify) <
                        since(VisionSystem(VisionSystem_GRC)(s2)=process)
```

3. The VisionSystem enters the state *alert*, internal event *Known* after incoming event *RecogC* or *RecogD*.

```
VisionSystem(VisionSystem_GRC)(s0) = alert ->
    since(VisionSystem(VisionSystem_GRC)(s1)=identify) <
                        since(VisionSystem(VisionSystem_GRC)(s2)=process)
```

5.2.2 Time constraint axioms

Time Constraint Axiom for RightArm

1. The RightArm sends the *Pick* event within 4 time units after receiving *RecogC* or *RecogD*.

```
RightArm(RightArm_GRC)(s0)=taken ->
    since(RightArm(RightArm_GRC)(s1)=ready) - since(RightArm(RightArm_GRC)(s2)=position) > 0
RightArm(RightArm_GRC)(s0)=taken ->
    since(RightArm(RightArm_GRC)(s1)=ready) - since(RightArm(RightArm_GRC)(s2)=position) > 4
```

2. The RightArm sends the *Assemble* event within 2 time units after receiving *Pick*

```
RightArm(RightArm_GRC)(s0)=finish ->
    since(RightArm(RightArm_GRC)(s1)=position) - since(RightArm(RightArm_GRC)(s2)=taken) > 0
RightArm(RightArm_GRC)(s0)=finish ->
    since(RightArm(RightArm_GRC)(s1)=position) - since(RightArm(RightArm_GRC)(s2)=taken) < 2
```

3. The RightArm sends the *PushC* or *PushD* event within 2 time units after receiving *Pick*

```
RightArm(RightArm_GRC)(s0)=ready ->
    since(RightArm(RightArm_GRC)(s1)=position) - since(RightArm(RightArm_GRC)(s2)=taken) > 0
RightArm(RightArm_GRC)(s0)=ready ->
    since(RightArm(RightArm_GRC)(s1)=position) - since(RightArm(RightArm_GRC)(s2)=taken) < 2
```

4. The RightArm sends the *Place* event within 2 time units after sending *Assemble*

```
RightArm(RightArm_GRC)(s0)=wait ->
    since(RightArm(RightArm_GRC)(s1)=taken) - since(RightArm(RightArm_GRC)(s2)=finish) > 0
RightArm(RightArm_GRC)(s0)=wait ->
    since(RightArm(RightArm_GRC)(s1)=taken) - since(RightArm(RightArm_GRC)(s2)=finish) < 2
```

Time constraint axioms for LeftArm

1. The LeftArm sends the *Pick* event within 4 time units after receiving *RecogC* or *RecogD*.

```
LeftArm(LeftArm_GRC)(s0)=taken ->
    since(LeftArm(LeftArm_GRC)(s1)=ready) - since(LeftArm(LeftArm_GRC)(s2)=position) > 0
LeftArm(LeftArm_GRC)(s0)=taken ->
    since(LeftArm(LeftArm_GRC)(s1)=ready) - since(LeftArm(LeftArm_GRC)(s2)=position) < 4
```

2. The LeftArm sends the *SynC* or *SynD* event within 1 time units after sending the *Pick* event.

```
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=position) - since(LeftArm(LeftArm_GRC)(s2)=taken) > 0
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=position) - since(LeftArm(LeftArm_GRC)(s2)=taken) < 1
```

3. The LeftArm sends the *SynC* event within 1 time units after sending the *PopC* event or sends the *SynD* event within 1 time units after sending the *PopD* event.

```
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=check) - since(LeftArm(LeftArm_GRC)(s2)=taken) > 0
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=check) - since(LeftArm(LeftArm_GRC)(s2)=taken) < 1
```

4. The LeftArm has internal event *Free* event within 2 time units after sending the *Assemble* event.

```
LeftArm(LeftArm_GRC)(s0)=check ->
    since(LeftArm(LeftArm_GRC)(s1)=wait) - since(LeftArm(LeftArm_GRC)(s2)=finish) > 0
LeftArm(LeftArm_GRC)(s0)=check ->
    since(LeftArm(LeftArm_GRC)(s1)=wait) - since(LeftArm(LeftArm_GRC)(s2)=finish) < 2
```

5. The LeftArm sends event *IsEmpty* event within 2 time units after internal event *Free*.

```
LeftArm(LeftArm_GRC)(s0)=check ->
    since(LeftArm(LeftArm_GRC)(s1)=wait) - since(LeftArm(LeftArm_GRC)(s2)=finish) > 0
LeftArm(LeftArm_GRC)(s0)=check ->
    since(LeftArm(LeftArm_GRC)(s1)=wait) - since(LeftArm(LeftArm_GRC)(s2)=finish) < 2
```

6. The LeftArm sends event *PopC* or *PopD* event within 2 time units after internal event *Free*.

```
LeftArm(LeftArm_GRC)(s0)=taken ->
   since(LeftArm(LeftArm_GRC)(s1)=finish) - since(LeftArm(LeftArm_GRC)(s2)=check) > 0
LeftArm(LeftArm_GRC)(s0)=taken ->
   since(LeftArm(LeftArm_GRC)(s1)=finish) - since(LeftArm(LeftArm_GRC)(s2)=check) < 2
```

Time constraint axioms for VisionSystem

1. The VisionSystem has internal event *Known* within 3 time units after the *Sensed* event.

```
VisionSystem(VisionSystem_GRC)(s0)=identify ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) -
       since(VisionSystem(VisionSystem_GRC)(s2)=process) > 0
VisionSystem(VisionSystem_GRC)(s0)=identify ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) -
       since(VisionSystem(VisionSystem_GRC)(s2)=process) < 3
```

2. The VisionSystem has internal event *Unknown* between 2 to 4 time units after the *Sensed* event.
   This time constraint cannot be expressed with the *since* operator. *since* does not include the concepts of period therefore including in one logical assertion $since(A = S_1)$ of a previous period and $since(A = S_1)$ of the current period cannot be done. Hence the following axioms are not conclusive and are to be excluded from the set of axioms.

```
VisionSystem(VisionSystem_GRC)(s0)=alert ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) -
       since(VisionSystem(VisionSystem_GRC)(s2)=process) > 2
VisionSystem(VisionSystem_GRC)(s0)=alert ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) -
       since(VisionSystem(VisionSystem_GRC)(s2)=process) < 4
```

3. The VisionSystem sends event *RecogC* or *RecogD* within 6 time units after the *Sensed* event.

```
VisionSystem(VisionSystem_GRC)(s0)=process ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) < 6
VisionSystem(VisionSystem_GRC)(s0)=identify ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) < 6
```

Time constraint axioms for Belt

1. The Belt has internal event *Stop* within 4 time units after the event *Sensed*.

```
Belt(Belt_GRC)(s0)=stopped -> since(Belt(Belt_GRC)(s1)=active) - since(Belt(Belt_GRC)(s2)=slow) > 0
Belt(Belt_GRC)(s0)=stopped -> since(Belt(Belt_GRC)(s1)=active) - since(Belt(Belt_GRC)(s2)=slow) < 4
```

2. The Belt has internal event *Move* between 5 and 7 time units after the internal event *Stop*.

```
Belt(Belt_GRC)(s0)=active -> since(Belt(Belt_GRC)(s1)=slow) - since(Belt(Belt_GRC)(s2)=stopped) > 5
Belt(Belt_GRC)(s0)=active -> since(Belt(Belt_GRC)(s1)=slow) - since(Belt(Belt_GRC)(s2)=stopped) < 7
```

5.2.3 Synchrony axioms

As the Figure 34 depicts, the scs includes a configuration list that can also be described axiomatically with the *since* operator. Each configuration line is a portlink which shows that the object instances communicate through the association. This section will enumerate the axioms as extracted by the theory in Chapter 3

1. When the event *Sensed* occurs in objects VisionSystem VS1 and in object Belt B1, VS1 comes in state *process* and B1 comes in state *slow*. Therefore the time since VS1 was in state *alert* is equal to the time since B1 was in state *active*.

```
( (VisionSystem(VS1)(s0)=process) ) ^ ( (Belt(B1)(s0)=slow) ) ->
      since(VisionSystem(VS1)(s1)=alert) = since(Belt(B1)(s2)=active)
```

2. When the event *Pick* occurs in objects RightArm RA1 and in object Belt B1, RA1 comes in state *taken* and B1 comes in state *stopped*. Therefore the time since RA1 was in state *position* is equal to the time since B1 was in state *stopped*.

```
No Axiom
```

3. When the events *PopC* and *PopD* occur in objects StackStore ST1 and in object LeftArm LA1, ST1 comes in state *active* and LA1 comes in state *taken*. Therefore the time since ST1 was in state *active* is not equal to the time since LA1 was in state *check*.

```
No axiom
```

4. When the event *IsEmpty* occurs in objects StackStore ST1 and in object LeftArm LA1, ST1 comes in state *active* and LA1 comes in state *ready*. Therefore the time since ST1 was in state *active* is not equal to the time since LA1 was in state *check*.

```
No axiom
```

5. When the events *RecogC* and *RecogD* occur in objects VisionSystem VS1 and in object LeftArm LA1, VS1 comes in state *alert* and LA1 comes in state *position*. Therefore the time since VS1 was in state *identify* is equal to the time since LA1 was in state *ready*.

```
( (VisionSystem(VS1)(s0)=alert) ) ^ ( (LeftArm(LA1)(s0)=position) ) ->
      since(VisionSystem(VS1)(s1)=identify) = since(LeftArm(LA1)(s2)=ready)
```

6. When the events *RecogC* and *RecogD* occur in objects VisionSystem VS1 and in object RightArm RA1, VS1 comes in state *alert* and RA1 comes in state *position*. Therefore the time since VS1 was in state *identify* is equal to the time since RA1 was in state *ready*.

```
( (VisionSystem(VS1)(s0)=alert) ) ^ ( (RightArm(RA1)(s0)=position) ) ->
      since(VisionSystem(VS1)(s1)=identify) = since(RightArm(RA1)(s2)=ready)
```

7. When the events *SynC* and *SynD* occur in objects LeftArm LA1 and in object RightArm RA1, LA1 comes in state *active* and RA1 comes in state *taken*. Therefore the time since LA1 was in state *taken* is equal to the time since RA1 was in state *wait*.

```
( (LeftArm(LA1)(s0)=wait) ) ^ ( (RightArm(RA1)(s0)=ready) ) ->
      since(LeftArm(LA1)(s1)=taken) = since(RightArm(RA1)(s2)=wait)
```

8. When the event *Assembles* occurs in objects LeftArm LA1 and in object RightArm RA1, LA1 comes in state *finish* and RA1 comes in state *finish*. Therefore the time since LA1 was in state *wait* is equal to the time since RA1 was in state *taken*.

```
( (LeftArm(LA1)(s0)=finish) ) ^ ( (RightArm(RA1)(s0)=finish) ) ->
      since(LeftArm(LA1)(s1)=wait) = since(RightArm(RA1)(s2)=taken)
```

9. When the events *PushC* and *PushD* occur in objects StackStore ST1 and in object RightArm RA1, ST1 comes in state *active* and RA1 comes in state *ready*. Therefore the time since ST1 was in state *active* is not equal to the time since RA1 was in state *taken*.

```
No axiom
```

10. When the event *Place* occurs in objects Tray TR1 and in object RightArm RA1, TR1 comes in state *wait* and RA1 comes in state *wait*. Therefore the time since TR1 was in state *wait* is not equal to the time since RA1 was in state *finish*.

```
No axiom
```

11. When the event Pick occurs in objects StackStore Belt B1 and in object LeftArm LA1, B1 comes in state stopped and LA1 comes in state taken. Therefore the time since B1 was in state stopped is not equal to the time since LA1 was in state position.

```
No axiom
```

5.2.4 Generated *since* axioms

```
Transition axioms for Tray

Transition axioms for StackStore

Transition axioms for RightArm

RightArm(RightArm_GRC)(s0) = ready ->
    since(RightArm(RightArm_GRC)(s1)=wait) < since(RightArm(RightArm_GRC)(s2)=finish)
RightArm(RightArm_GRC)(s0) = position ->
    since(RightArm(RightArm_GRC)(s1)=ready) < since(RightArm(RightArm_GRC)(s2)=wait)
RightArm(RightArm_GRC)(s0) = position ->
    since(RightArm(RightArm_GRC)(s1)=ready) < since(RightArm(RightArm_GRC)(s2)=taken)
RightArm(RightArm_GRC)(s0) = wait ->
    since(RightArm(RightArm_GRC)(s1)=finish) < since(RightArm(RightArm_GRC)(s2)=taken)
RightArm(RightArm_GRC)(s0) = taken ->
    since(RightArm(RightArm_GRC)(s1)=position) < since(RightArm(RightArm_GRC)(s2)=ready)
RightArm(RightArm_GRC)(s0) = ready ->
    since(RightArm(RightArm_GRC)(s1)=taken) < since(RightArm(RightArm_GRC)(s2)=position)
RightArm(RightArm_GRC)(s0) = ready ->
   since(RightArm(RightArm_GRC)(s1)=taken) < since(RightArm(RightArm_GRC)(s2)=position)
RightArm(RightArm_GRC)(s0) = finish ->
   since(RightArm(RightArm_GRC)(s1)=taken) < since(RightArm(RightArm_GRC)(s2)=position)


Transition axioms for LeftArm

LeftArm(LeftArm_GRC)(s0) = taken ->
   since(LeftArm(LeftArm_GRC)(s1)=position) < since(LeftArm(LeftArm_GRC)(s2)=ready)
LeftArm(LeftArm_GRC)(s0) = wait ->
   since(LeftArm(LeftArm_GRC)(s1)=taken) < since(LeftArm(LeftArm_GRC)(s2)=position)
LeftArm(LeftArm_GRC)(s0) = wait ->
   since(LeftArm(LeftArm_GRC)(s1)=taken) < since(LeftArm(LeftArm_GRC)(s2)=check)
LeftArm(LeftArm_GRC)(s0) = position ->
   since(LeftArm(LeftArm_GRC)(s1)=ready) < since(LeftArm(LeftArm_GRC)(s2)=check)
LeftArm(LeftArm_GRC)(s0) = finish ->
   since(LeftArm(LeftArm_GRC)(s1)=wait) < since(LeftArm(LeftArm_GRC)(s2)=taken)
LeftArm(LeftArm_GRC)(s0) = taken ->
   since(LeftArm(LeftArm_GRC)(s1)=check) < since(LeftArm(LeftArm_GRC)(s2)=finish)
LeftArm(LeftArm_GRC)(s0) = ready ->
   since(LeftArm(LeftArm_GRC)(s1)=check) < since(LeftArm(LeftArm_GRC)(s2)=finish)
LeftArm(LeftArm_GRC)(s0) = check ->
   since(LeftArm(LeftArm_GRC)(s1)=finish) < since(LeftArm(LeftArm_GRC)(s2)=wait)
```

```
Transition axioms for VisionSystem

VisionSystem(VisionSystem_GRC)(s0) = identify ->
    since(VisionSystem(VisionSystem_GRC)(s1)=process) < since(VisionSystem(VisionSystem_GRC)(s2)=alert)
VisionSystem(VisionSystem_GRC)(s0) = alert ->
    since(VisionSystem(VisionSystem_GRC)(s1)=identify) < since(VisionSystem(VisionSystem_GRC)(s2)=process)
VisionSystem(VisionSystem_GRC)(s0) = alert ->
    since(VisionSystem(VisionSystem_GRC)(s1)=identify) < since(VisionSystem(VisionSystem_GRC)(s2)=process)
VisionSystem(VisionSystem_GRC)(s0) = process ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) < since(VisionSystem(VisionSystem_GRC)(s2)=identify)
VisionSystem(VisionSystem_GRC)(s0) = process ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) < since(VisionSystem(VisionSystem_GRC)(s2)=identify)


Transition axioms for Belt

Belt(Belt_GRC)(s0) = stopped ->
    since(Belt(Belt_GRC)(s1)=slow) < since(Belt(Belt_GRC)(s2)=active)
Belt(Belt_GRC)(s0) = active ->
    since(Belt(Belt_GRC)(s1)=stopped) < since(Belt(Belt_GRC)(s2)=slow)
Belt(Belt_GRC)(s0) = slow ->
    since(Belt(Belt_GRC)(s1)=active) < since(Belt(Belt_GRC)(s2)=stopped)

TC Axiom : TCvar1 of RightArm.
RightArm(RightArm_GRC)(s0)=taken ->
    since(RightArm(RightArm_GRC)(s1)=ready) - since(RightArm(RightArm_GRC)(s2)=position) > 0
RightArm(RightArm_GRC)(s0)=taken ->
    since(RightArm(RightArm_GRC)(s1)=ready) - since(RightArm(RightArm_GRC)(s2)=position) < 4

TC Axiom : TCvar2 of RightArm.
RightArm(RightArm_GRC)(s0)=taken ->
    since(RightArm(RightArm_GRC)(s1)=ready) - since(RightArm(RightArm_GRC)(s2)=position) > 0
RightArm(RightArm_GRC)(s0)=taken ->
    since(RightArm(RightArm_GRC)(s1)=ready) - since(RightArm(RightArm_GRC)(s2)=position) < 4

TC Axiom : TCvar3 of RightArm.
RightArm(RightArm_GRC)(s0)=finish ->
    since(RightArm(RightArm_GRC)(s1)=position) - since(RightArm(RightArm_GRC)(s2)=taken) > 0
RightArm(RightArm_GRC)(s0)=finish ->
    since(RightArm(RightArm_GRC)(s1)=position) - since(RightArm(RightArm_GRC)(s2)=taken) < 2

TC Axiom : TCvar4 of RightArm.
RightArm(RightArm_GRC)(s0)=ready ->
    since(RightArm(RightArm_GRC)(s1)=position) - since(RightArm(RightArm_GRC)(s2)=taken) > 0
RightArm(RightArm_GRC)(s0)=ready ->
    since(RightArm(RightArm_GRC)(s1)=position) - since(RightArm(RightArm_GRC)(s2)=taken) < 2
```

```
TC Axiom : TCvar5 of RightArm.
RightArm(RightArm_GRC)(s0)=ready ->
    since(RightArm(RightArm_GRC)(s1)=position) - since(RightArm(RightArm_GRC)(s2)=taken) > 0
RightArm(RightArm_GRC)(s0)=ready ->
    since(RightArm(RightArm_GRC)(s1)=position) - since(RightArm(RightArm_GRC)(s2)=taken) < 2

TC Axiom : TCvar6 of RightArm.
RightArm(RightArm_GRC)(s0)=wait ->
    since(RightArm(RightArm_GRC)(s1)=taken) - since(RightArm(RightArm_GRC)(s2)=finish) > 0
RightArm(RightArm_GRC)(s0)=wait ->
    since(RightArm(RightArm_GRC)(s1)=taken) - since(RightArm(RightArm_GRC)(s2)=finish) < 2

TC Axiom : TCvar1 of VisionSystem.
VisionSystem(VisionSystem_GRC)(s0)=identify ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) - since(VisionSystem(VisionSystem_GRC)(s2)=process) > 0
VisionSystem(VisionSystem_GRC)(s0)=identify ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) - since(VisionSystem(VisionSystem_GRC)(s2)=process) < 3

TC Axiom : TCvar2 of VisionSystem.
VisionSystem(VisionSystem_GRC)(s0)=alert ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) - since(VisionSystem(VisionSystem_GRC)(s2)=process) > 2
VisionSystem(VisionSystem_GRC)(s0)=alert ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) - since(VisionSystem(VisionSystem_GRC)(s2)=process) < 4

TC Axiom : TCvar3 of VisionSystem.
VisionSystem(VisionSystem_GRC)(s0)=process ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) < 6
VisionSystem(VisionSystem_GRC)(s0)=identify ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) < 6
VisionSystem(VisionSystem_GRC)(s0)=alert ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) < 6

TC Axiom : TCvar4 of VisionSystem.
VisionSystem(VisionSystem_GRC)(s0)=process ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) < 6
VisionSystem(VisionSystem_GRC)(s0)=identify ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) < 6
VisionSystem(VisionSystem_GRC)(s0)=alert ->
    since(VisionSystem(VisionSystem_GRC)(s1)=alert) < 6

TC Axiom : TCvar1 of Belt.
Belt(Belt_GRC)(s0)=stopped ->
    since(Belt(Belt_GRC)(s1)=active) - since(Belt(Belt_GRC)(s2)=slow) > 0
Belt(Belt_GRC)(s0)=stopped ->
    since(Belt(Belt_GRC)(s1)=active) - since(Belt(Belt_GRC)(s2)=slow) < 4
```

```
TC Axiom : TCvar2 of Belt.
Belt(Belt_GRC)(s0)=active ->
   since(Belt(Belt_GRC)(s1)=slow) - since(Belt(Belt_GRC)(s2)=stopped) > 5
Belt(Belt_GRC)(s0)=active ->
   since(Belt(Belt_GRC)(s1)=slow) - since(Belt(Belt_GRC)(s2)=stopped) < 7

TC Axiom : TCvar1 of LeftArm.
LeftArm(LeftArm_GRC)(s0)=taken ->
    since(LeftArm(LeftArm_GRC)(s1)=ready) - since(LeftArm(LeftArm_GRC)(s2)=position) > 0
LeftArm(LeftArm_GRC)(s0)=taken ->
    since(LeftArm(LeftArm_GRC)(s1)=ready) - since(LeftArm(LeftArm_GRC)(s2)=position) < 4

TC Axiom : TCvar2 of LeftArm.
LeftArm(LeftArm_GRC)(s0)=taken ->
    since(LeftArm(LeftArm_GRC)(s1)=ready) - since(LeftArm(LeftArm_GRC)(s2)=position) > 0
LeftArm(LeftArm_GRC)(s0)=taken ->
    since(LeftArm(LeftArm_GRC)(s1)=ready) - since(LeftArm(LeftArm_GRC)(s2)=position) < 4

TC Axiom : TCvar3 of LeftArm.
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=position) - since(LeftArm(LeftArm_GRC)(s2)=taken) > 0
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=position) - since(LeftArm(LeftArm_GRC)(s2)=taken) < 1

TC Axiom : TCvar4 of LeftArm.
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=position) - since(LeftArm(LeftArm_GRC)(s2)=taken) > 0
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=position) - since(LeftArm(LeftArm_GRC)(s2)=taken) < 1

TC Axiom : TCvar5 of LeftArm.
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=check) - since(LeftArm(LeftArm_GRC)(s2)=taken) > 0
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=check) - since(LeftArm(LeftArm_GRC)(s2)=taken) < 1

TC Axiom : TCvar6 of LeftArm.
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=check) - since(LeftArm(LeftArm_GRC)(s2)=taken) > 0
LeftArm(LeftArm_GRC)(s0)=wait ->
    since(LeftArm(LeftArm_GRC)(s1)=check) - since(LeftArm(LeftArm_GRC)(s2)=taken) < 1

TC Axiom : TCvar7 of LeftArm.
LeftArm(LeftArm_GRC)(s0)=check ->
    since(LeftArm(LeftArm_GRC)(s1)=wait) - since(LeftArm(LeftArm_GRC)(s2)=finish) > 0
LeftArm(LeftArm_GRC)(s0)=check ->
    since(LeftArm(LeftArm_GRC)(s1)=wait) - since(LeftArm(LeftArm_GRC)(s2)=finish) < 2
```

```
TC Axiom : TCvar8 of LeftArm.
LeftArm(LeftArm_GRC)(s0)=ready ->
    since(LeftArm(LeftArm_GRC)(s1)=finish) - since(LeftArm(LeftArm_GRC)(s2)=check) > 0
LeftArm(LeftArm_GRC)(s0)=ready ->
    since(LeftArm(LeftArm_GRC)(s1)=finish) - since(LeftArm(LeftArm_GRC)(s2)=check) < 2

TC Axiom : TCvar9 of LeftArm.
LeftArm(LeftArm_GRC)(s0)=taken ->
    since(LeftArm(LeftArm_GRC)(s1)=finish) - since(LeftArm(LeftArm_GRC)(s2)=check) > 0
LeftArm(LeftArm_GRC)(s0)=taken ->
    since(LeftArm(LeftArm_GRC)(s1)=finish) - since(LeftArm(LeftArm_GRC)(s2)=check) < 2

TC Axiom : TCvar10 of LeftArm.
LeftArm(LeftArm_GRC)(s0)=taken ->
    since(LeftArm(LeftArm_GRC)(s1)=finish) - since(LeftArm(LeftArm_GRC)(s2)=check) > 0
LeftArm(LeftArm_GRC)(s0)=taken ->
    since(LeftArm(LeftArm_GRC)(s1)=finish) - since(LeftArm(LeftArm_GRC)(s2)=check) < 2

Configuration line # 1
( (VisionSystem(VS2)(s0)=process) ) ^ ( (Belt(B2)(s0)=slow) ) ->
      since(VisionSystem(VS2)(s1)=alert) = since(Belt(B2)(s2)=active)

Configuration line # 2
( (RightArm(RA2)(s0)=taken) ) ^ ( (Belt(B2)(s0)=stopped) ) ->
      since(RightArm(RA2)(s1)=position) = since(Belt(B2)(s2)=stopped)
The preceding axiom is an INVALID AXIOM


Configuration line # 3
( (StackStore(ST2)(s0)=Active) ) ^ ( (LeftArm(LA2)(s0)=taken) ) ->
      since(StackStore(ST2)(s1)=Active) = since(LeftArm(LA2)(s2)=check)
The preceding axiom is an INVALID AXIOM


( (StackStore(ST2)(s0)=Active) ) ^ ( (LeftArm(LA2)(s0)=taken) ) ->
      since(StackStore(ST2)(s1)=Active) = since(LeftArm(LA2)(s2)=check)
The preceding axiom is an INVALID AXIOM


( (StackStore(ST2)(s0)=Active) ) ^ ( (LeftArm(LA2)(s0)=ready) ) ->
      since(StackStore(ST2)(s1)=Active) = since(LeftArm(LA2)(s2)=check)
The preceding axiom is an INVALID AXIOM


Configuration line # 4
( (VisionSystem(VS2)(s0)=alert) ) ^ ( (LeftArm(LA2)(s0)=position) ) ->
      since(VisionSystem(VS2)(s1)=identify) = since(LeftArm(LA2)(s2)=ready)
```

```
( (VisionSystem(VS2)(s0)=alert) ) ^ ( (LeftArm(LA2)(s0)=position) ) ->
      since(VisionSystem(VS2)(s1)=identify) = since(LeftArm(LA2)(s2)=ready)

Configuration line # 5
( (VisionSystem(VS2)(s0)=alert) ) ^ ( (RightArm(RA2)(s0)=position) ) ->
      since(VisionSystem(VS2)(s1)=identify) = since(RightArm(RA2)(s2)=ready)

( (VisionSystem(VS2)(s0)=alert) ) ^ ( (RightArm(RA2)(s0)=position) ) ->
      since(VisionSystem(VS2)(s1)=identify) = since(RightArm(RA2)(s2)=ready)

Configuration line # 6
( (LeftArm(LA2)(s0)=wait) ) ^ ( (RightArm(RA2)(s0)=ready) ) ->
      since(LeftArm(LA2)(s1)=taken) = since(RightArm(RA2)(s2)=wait)

( (LeftArm(LA2)(s0)=wait) ) ^ ( (RightArm(RA2)(s0)=ready) ) ->
      since(LeftArm(LA2)(s1)=taken) = since(RightArm(RA2)(s2)=wait)

( (LeftArm(LA2)(s0)=finish) ) ^ ( (RightArm(RA2)(s0)=finish) ) ->
      since(LeftArm(LA2)(s1)=wait) = since(RightArm(RA2)(s2)=taken)

Configuration line # 7
( (StackStore(ST2)(s0)=Active) ) ^ ( (RightArm(RA2)(s0)=ready) ) ->
      since(StackStore(ST2)(s1)=Active) = since(RightArm(RA2)(s2)=taken)
The preceding axiom is an INVALID AXIOM


( (StackStore(ST2)(s0)=Active) ) ^ ( (RightArm(RA2)(s0)=ready) ) ->
      since(StackStore(ST2)(s1)=Active) = since(RightArm(RA2)(s2)=taken)
The preceding axiom is an INVALID AXIOM


Configuration line # 8
( (Tray(TR2)(s0)=Wait) ) ^ ( (RightArm(RA2)(s0)=wait) ) ->
      since(Tray(TR2)(s1)=Wait) = since(RightArm(RA2)(s2)=finish)
The preceding axiom is an INVALID AXIOM


Configuration line # 9
( (Belt(B2)(s0)=stopped) ) ^ ( (LeftArm(LA2)(s0)=taken) ) ->
      since(Belt(B2)(s1)=stopped) = since(LeftArm(LA2)(s2)=position)
The preceding axiom is an INVALID AXIOM
```

5.2.5 Commenting *since* generated output

This section shows the differences between the manually derived *since* axioms and the automatically derived axioms. Some axioms will be identified and reasons explaining the differences, hence the limitations of the tool will be given.

The transition axioms and the time-constraint axioms respect the manually generated axioms.

The generated set of synchrony axioms have some axioms identified as invalid instead of simply being suppressed. The reason for this is that the algorithm that generates the synchrony axioms treats the information incrementally and outputs as it goes through the AST. Moreover, the characteristics that identify the axiom as being invalid is scanned late in the algorithmic process. We therefore have an axiom output and a comment is added afterwards. The axiom identified by the lines 205 to 208 is an example.

Also in the synchrony axioms, the axioms shown on lines 238 to 243 are repeated. This is due to the fact that some transitions are doubled by the cup or dish duality. That is, going from state S_1 to S_2 can occur through two different transitions triggered by two different events, in this case the events *RecogC* and *RecogD*.

Chapter 6

Conclusion

6.1 Work synthesis

The flow of this thesis can be briefly resumed with the following items:

- Chapter 1 introduces the high level concepts of the field of this thesis' work which are reactive systems, formal methods and formal verifications.
- Chapter 2 introduces the model with which this thesis is to work with, the TROM model. Its formalism, attributes and characteristics are presented.
- Chapter 4 presents the design of the axiomatic description generator tool. The PVS generator and the *since* expression generator are shown. The requirements, the associated algorithms to solve the main problems and the structure of the tool is presented.
- Finally, Chapter 5 presents a new case for the application of the deriving methodology described in [MA99] and the associated application of both tools, the PVS generator and the *since* generator.

Chapter 5 also presents commented results of the tool. Since limitations do exist, they have been highlighted and described.

With the development of the TROM axiomatic description generator described in this thesis, TROMLAB users now have the grounds set for a complete mechanically assisted prototype development cycle. The tools along the development cycle are:

- The UML **TROM** model for UML graphic based **TROM** specifications [AM99]
- The UML-ROSE translator that brings the UML **TROM** specifications to the original **TROM** formal semantics [Pop99]
- The **TROM** interpreter that parses and creates the AST internal structure [Tao96, Sri99]
- The **TROMLAB** simulator to validate models [Mut96]
- The reasoning system to further enhance model validation [Hai99]
- The axiomatic description generator to execute translation to a mechanical proof tool described in this thesis
- The PVS tool to use for its theorem prover [Sha92]

6.2 Future work

As we saw through the tool specifications and through the case study, one of the current challenges to a more complete set of rules for automatically axiomatizing the TROM model, is the complexity induced by cycles. This limitation can also be applied to any algorithm trying to grasp specifications modeled on state machines. We saw that the relationships between occurrences of events can often be quickly understood with an intuitive analysis but trying to create algorithms to derive the factual data from models it became evident that a deeper analysis would be required. Therefore, an area for future works that would bring a lot of benefits to the formal specifications community would be cycle analysis of state machine models.

By having such limitations, we see that the current level of the developed tool is still at an *assistant* stage. The day of a fully automated axiomatizing tool, where minimal intervention from model designers would be required is still a few research iterations away. Nonetheless, we must remain optimistic towards the available results that are provided by formal specifications which enable the creation of safer, more reliable and better quality software.

Bibliography

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

[AAR95a] R. Achuthan, V. S. Alagar, and T. Radhakrishnan. An object-oriented modeling of real-time robotic assembly system. In *Proceedings of IEEE First International Conference on Engineering of Complex Computer Systems, ICECCS'95*, Florida, October 1995.

[AAR95b] R. Achuthan, V. S. Alagar, and T. Radhakrishnan. TROM - an object model for reactive system development. In *The 1995 Asian Computing Science Conference, ASIAN'95*, Thailand, December 1995.

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

[AM] V.S. Alagar and D. Muthiayen. Notes from comp748. Course notes for Comp748, Concordia University, Montréal, Canada.

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

[AM99] V. S. Alagar and D. Muthiayen. A formal approach to uml modeling of complex real-time reactive systems. Submitted for publication in *IEEE Transactions on Software Engineering*, July 1999.

[AMP99] V. S. Alagar, D. Muthiayen, and F. Pompeo. From behavioral specifications to axiomatic description of real-time reactive systems. In *Proceedings of Fifth IEEE Real-Time Technology and Application Symposium Work-in-Progress Session, RTAS'99 WIP, Vancouver, Canada*, June 1999.

[BM88] R. Boyer and J. Moore. *A Computational Logic Handbook*. Academic Press, New-York, 1988.

[Bol96] T. Bolognesi. Constraint-oriented specifications style for time-dependent behaviours. In *Formal Methods for Real-Time Computing*, pages 195–202. John Wiley & sons, 1996.

[CES86] M Clarke, E Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systens*, 1986.

[CM92] J. Carruth and J. Misra. Proof of real-time mutual-exclusion algorithm. Notes on UNITY, 1992.

[COR$^+$95] J. Crow, S. Owre, J Rushby, N Shankar, and S. Mandayam. A tutorial introduction to pvs. In *Workshop on Industrial-Strenght Formal Specification Techniques, Boca Raton, Florida*. SRI International, 1995.

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

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

[HD96] C. Heitmeyer and Mandrioli D. Formal methods for real-time computing: An overview. In *Formal Methods for Real-Time Computing*, pages 1–32. John Wiley & sons, 1996.

[MA99] D. Muthiayen and V.S. Alagar. Mechanized verification of real-time reactive systems in an object-oriented framework. Submitted to IEEE Software Transactions on Software Engineering, 1999.

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

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

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

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

[Sha92] N. Shankar. Mechanized verification of real-time systems using pvs. Technical report, SRI, 1992.

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

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

Appendix A

Class dictionary for Generator tool

- **Since_main** - This class is the controlling class. It the main called where the AST building operator is called and then it creates the Since_generator class and calls its *run*() routine.
- **Since_Generator** - This class is where the bulk of the work is. It creates the necessary objects and then calls the axiom generating algorithms.
- **Since_Statelist** - This class is an extension of the List class. It it used to maintain lists of states used in the Since_generator class.
- **Since_State** - This class defines the state object used in Since_Generator.
- **PVS_main** - This class is the controlling class. It the main called where the AST building operator is called and then it creates the PVS_generator class and calls its *run*() routine.
- **PVS_Event** - Class representing the event type needed for the axiom generator.
- **PVS_setup** - Class that contains routines for the setup of the PVS theories, files
- **PVS_Eventlist** - List of PVS_Event, an extension from the List class
- **PVS_Generator** - This class is where the bulk of the work is. It creates the necessary objects and then calls the axiom generating algorithms.

Table 3: *Since_state* attributes

Attribute	*Type*	*definition*
state_name	String	State name
is_initial	boolean	if state is an initial state, is_initial is true
visited	boolean	to be used in graph search algorithm, if the state is visited while graph being searched, visited is set to true
substate_list	Since_statelist	if state is a complex state the list of its substates

Table 4: *Since_Generator* attributes

Attribute	*Type*	*definition*
trom_asts	Tromclasslist	List of TROMs in the built AST
scs_asts	SCSlist	Lists of SCSs in the AST at hand

Table 5: *Since_Generator* operations

Operation	note	returns	parameters
Since_Generator	Constructor	Since_Generator	TromClasslist, SCSList
run	Main routine of the since Generator. It calls all the axiom generating algorithms	void	nil
generate_tr_since	This routine is the main algorithm to extract the transition axioms from the AST and execute the transformation to obtain *since* expressions	void	nil
generate_tc_since	This routine is the main algorithm to extract the time-constraint axioms from the AST and execute the transformation to obtain *since* expressions	void	nil
generate_syn_Since	This routine is the main algorithm to extract the synchrony axioms from the AST and execute the transformation to obtain *since* expression	void	nil
create_statelist-_from_since_statelist	This routine creates a copy of a Since_statelist object list object as opposed to creating a reference only	Since_statelist	Since_statelist
create_statelist	Creates a since state list from a TROM state list	Since_statelist	statelist
display_trans-_spec_Since	Displays the transition axioms	void	Int, Node (ts) Node(ts), Node(TROM
display_time-_constraint_Since	Displays the time const. axioms, uses the between algo	void	Int, Node(ts), Node(ts) Node(TROM), Node (tc)
display_syn_Since	Displays the syn axioms	void objlabel1, objelabel2, Event	Int, tromname, tromname,
exists_path	returns if there's a path between two states	boolean Since_statelist, Since_statelist	trom, state1, state2,
next_state_list	returns states next to a state	Since_statelist visited bool	trom, state, Since_statelist,
between	returns statelist of states between two states	state list Since_statelist, int	stateA, stateB, trom
get_event_name-_from_port_type	returns an event from a port type	event name	port type name
get_trom_name-_from_obj_label	returns a trom name from an object label	trom name	object label

Table 6: *Since_state* operations

Operation	note	returns	parameters
Since_state	Constructor	Since_state	
set_visited	Sets the visited boolean to true	void	nil
been_visited	Checks the visited boolean and returns boolean accordingly	boolean	nil
state_name	Returns the state name	String	nil
is_initial	Returns whether the state is flagged as an initial state	boolean	nil
get_substate_list	returns the reference to the substate list	Since_statelist	nil

Table 7: *Since_statelist* operations

Operation	note	returns	parameters
Since_statelist	Constructor	Since_statelist	void
get_state	returns the state object	Since_state	state name
contains	Checks the list for the state name submitted in parameter and returns a boolean if it does find the state name	boolean	state name

Table 8: *PVS_Generator* and *PVS_Setup* attributes

Attribute	*Type*	*definition*
trom_asts	Tromclasslist	List of TROMs in the built AST
scs_asts	SCSlist	Lists of SCSs in the AST at hand

Table 9: *PVS_Event* attributes

Attribute	*Type*	*definition*
grc_name	String	GRC name
event_name	String	event name
port_type_name	String	port_type name

Table 10: *PVS_Generator* operations

Operation	note	returns	parameters
PVS_Generator	Constructor	PVS_Generator	TromClasslis, SCSList
run	Routine called by PVS_main to start the whole process	void	nil
Generate_theory	Routine that dispatches the theory generation and formatting	void	trom, setup object
generate_tr_axioms	generates the transition axioms	void	trom
generate_tc_axioms	generates the time-constraint axioms	void	trom
generate_syn_axioms	generates the synchrony axioms	void	nil
display_trans-_spec_axiom	executes the displaying	void	Int, ts1, ts2, evtlist, boolean
display_time-_constraint_axiom	executes the displaying	void	Int, tc, ts, trom
display_syn_axiom	executes the displaying	void	Int, name1, name2, obj, obj2, evt
generate_time_expression	routine that creates a string in the format of a time spec of PVS	void	trom, evt
create_event_list	creates a PVS_Event list from the GRC	PVS_Eventlist	nil
create_event-_list_for_trom	creates a PVS_Event list from the TROM	PVS_Eventlist	trom
get_event_name-_from_port_type	gets an event name allowed in a port type	String	grc, port type, event
get_trom_name-_from_obj_label	returns the TROM name from a label of the SCS	String	object

Table 11: *PVS_Event* operations

Operation	note	returns	parameters
PVS_Event	constructor	PVS_Event	nil
get_grcname	gets the name of the grc	String	nil
get_eventname	gets the name of the event	String	nil
get_port_type_name	gets the name of the port type	String	nil

Table 12: *PVS_Eventlist* operations

Operation	note	returns	parameters
PVS_Eventlist	constructor	PVS_Eventlist	nil
contains_event	returns whether the lists contains an event	boolean	event, grc
get_PVS_event	returns the requested event from the list	PVS_event	event, grc
append_PVS_Event	appends an event to the list	void	PVS_Event

Table 13: *PVS_setup* operations

Operation	note	returns	parameters
PVS_setup			
run			
generate_types_and_vars	outputs info	void	nil
generate_trom_related_info	outputs info	void	trom
generate_scs_related_info	outputs info	void	nil