AUTOMATIC GENERATION OF BEHAVIORAL SPECIFICATION IN AUTONOMIC SYSTEMS TIMED REACTIVE MODEL

JAVIER QUIROZ

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

AND

SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTERS OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

NOVEMBER 2007

Your file Votre référence
ISBN: 978-0-494-40951-0
Our file Notre référence
ISBN: 978-0-494-40951-0

Abstract

Automated Generation of Behavioral Specification in Autonomic Systems Timed Reactive Model

Javier Quiroz

The Autonomic Reactive System Timed Reactive Model (AS-TRM) is the merging of two fields: Real Time Reactive Systems and Autonomic Systems. Autonomic Systems is a new research area conceived to deal with the growing complexity of nowadays Information Technology infrastructures. Time Reactive Systems are complex systems that interact with their environment using a stimulus-response behavior under strict timing constraints. The AS-TRM approach consists in generating a new breed of systems provided with autonomic self-management capabilities. Timed Reactive Object Model (TROM) formalism developed at Concordia University has been extended to model the reactive behaviour in autonomic systems. This thesis work is aimed at assessing the control on the behavioral correctness of the Autonomic Reactive Component layer in AS-TRM. An algorithm for generating an exhaustive behavior of an autonomic reactive component is presented; the algorithm guarantees the correctness of the component's behavioural specification by building the correct behaviour, timing constraints and system policies into the generated output. The specification of such behaviour is applicable to a variety of self-monitoring purposes concerning the autonomic reactive system control. Additionally, a method for assessing the critical time performance (minimum and maximum time delay) from the behavioural specification is provided. The methodology is illustrated on a case study.

To my father

Acknowledgments

I want to express my acknowledgment and gratitude to my supervisor Dr. Olga Ormandjieva for her technical and financial support as well for her encouragement and experienced guidance which helped me throughout the completion of this research work. Also, thanks to Heng Kuang for his technical advice while working on this research project. Thanks to the members of my research lab for providing suggestions and feedback. Thanks to the National Counselling of Science and Technology of Mexico (CONACYT) who sponsored and financed this research work.

I want to thank to all the members of my family; without them, nothing of this would have been possible. Specially, I want to thank to my mother and father for all their love and support. Also, thanks to Ion and Florina Simionescue and their family for their support and friendship. Thanks to Dr. Nicole Kerschberg and her family for her support and friendship. A very special thanks goes to Phoebe Wolf for her support and love.

Table of Contents

List of Figures

List of Tables

Chapter1. Introduction

This chapter introduces the context of this research in autonomic computing and real-time reactive systems; the Timed Reactive Object Model (TROM) and the Autonomic System Timed Reactive Model (AS-TRM). Also, the motivations behind this research and the scope of this thesis are addressed.

1.1 Autonomic Computing

Autonomic Computing is a relatively new research area in Computer Science that emerged as an initiative by IBM in 2002 [1]. It was conceived as a way to deal with the growing complexity that today's large IT infrastructures represent.

Nowadays, more and more sophisticated services are available from online shopping stores to online banking accounts. The availability of these services poses great challenges to IT staffs who must maintain and manage all the components in the system's infrastructure. According to Klein [2], up to 80 percent of an average company's IT budget is spent in maintaining existing applications. Since most of the time of the IT staff is devoted to manage core business processes [3] there is little time to spot potential areas of growth. In order to enable companies to focus on the application of technology to new business opportunities and innovation, the IT industry must address this complexity [3]. The response of autonomic computing is to create autonomic systems capable of managing themselves by using technology to manage technology.

1.2 Real-Time Reactive Systems

Real Time Reactive Systems are systems which interact continuously with their environment through a stimulus/response mechanism which can last indefinitely. Real time reactive systems are rigorously regulated by time constraints in their behavior and are considered one of the most complex types of systems. Examples of such systems are air traffic controllers, online payment systems, avionics systems and others. An important characteristic of Real-Time Reactive Systems is that failure is not a viable option. Hence, all the system's properties regarding functionality and timing have to be ensured to satisfy the safety requirements before a system is deployed. In order to do that correctly, rigorous techniques have to be applied to modeling, design and analysis of a system behavior prior to implementation.

1.3 AS-TRM

AS-TRM [6] is a hybrid architecture that merges the real time reactive systems and autonomic systems paradigms. AS-TRM builds on Timed Reactive Object Model (TROM) [4] architecture and extends it to suit autonomic systems properties. TROM is a formalism created to the design and development of Real Time Reactive Systems. Based on object oriented principles, the formalism allows system modularity, compositionality and hierarchy. The formalism is made up of three tiers that hierarchically abstract complexity of the system design. The bottom tier uses the Larch Shared Language (LSL) [5] for defining abstract data types to be used by the middle tier. The middle tier consist of a generic reactive class that is an extended state machine with ports, timing constraints and logical assertions on the attributes. Interaction between generic reactive classes is made by message passing expressing system events on compatible ports. Finally, the top tier models the collaboration, interaction and communication of interacting generic reactive classes. A development framework: TROMLAB, has been created based on the TROM formalism for design and development of real-time reactive systems [4].

AS-TRM architecture consists of five tiers designed to the composition of self-managed, distributed, proactive and evolving autonomic real-time reactive systems [6]. At the moment of writing this thesis there is no other research work, to our knowledge, that combines the two areas of research into a single framework.

1.4 Motivation and Context

AS-TRM represents an open research area in expansion and considerable research work has to be done to develop its potential; this thesis is oriented to contribute to that research effort. Previous work in AS-TRM by Heng Kuang [6] demonstrated the use of a calculation methodology that provides the self-assessment of reliability in AS-TRM. This reliability calculation method, based on the Markov chains theory, was applied to a set of states of a group of reactive objects whose reliability had to be evaluated. The set of states to be analyzed was comprised by all the legal states and transitions that the group of reactive objects could produce by the use of external or internal events. The calculation technique to abstract all the possible states and transitions that several state machines can produce in conjunction is denominated synchronous product machine. Also, the set of all legal states and transitions is represented by the set of all possible states and transitions minus the undesired states that violates some system properties like safety or liveness. Important is to notice that all the legal states and transitions of a reactive system defines its legal behavior and that its correct composition represents the correctness of the system's behavior against its specification. Such information provides important insight useful for the development of real time reactive systems especially at design time. Additionally, a data repository loaded with the legal behavior of a group of reactive objects might be useful in assisting run-time self-monitoring. Having the previous considerations in mind, this thesis research work was directed to provide AS-TRM with automatic generation functionality of the legal behavior of a group of autonomic reactive objects. In order to do that, the following goals where set:

a) To modify and extend the AS-TRM formalism to suit the needs of the specification that a synchronous product machine represents in the context of AS-TRM.

b) Since the behavior of a reactive system is also governed by properties like safety and liveness; to design a specification to include such properties as part of the autonomic reactive component specification.

c) To design and implement an algorithm to produce automatically a synchronous product machine taking as input the modified and new specifications designed in a) and b).

1.5 Organization of the Thesis

- Chapter 2. The related research work is surveyed in this chapter.
- Chapter 3. This chapter introduces the TROM formalism, its notation and the TROMLAB framework.
- Chapter 4. The AS-TRM is described as an hybrid of both: Autonomic Systems and Real-Time Reactive System.

- Chapter 5. In this chapter, a detailed explanation of the algorithm for calculating the synchronous product machine is provided. The specifications of liveness and safety policies as well as other extensions to the AS-TRM current model are introduced. Finally, the minimum and maximum delay algorithm is presented.
- Chapter 6. This chapter shows the design and the implementation of the algorithm. Tests are conducted on the Railroad case study and the results are presented.
- Chapter 7. The conclusions are shown in this chapter amongst suggestions for future research directions.

Chapter 2. Related Work

This chapter reviews the research work in the AS-TRM area as well as in the automatic generation of synchronous product machine for real time reactive systems modeled in TROM. Also, the verification of the correctness of a system in the context of real-time reactive systems modeled with TROM formalism is surveyed here.

The research work reported in this thesis addresses the problem of self-monitoring the correctness of an evolving behavior in reactive autonomic systems modeled with the Autonomic System Timed Reactive Model (AS-TRM) [6][7][9][16][17][18][19], where each autonomic component is designed to react to every request from the environment in real time and synchronize its behavior with the environment (including other autonomic components). Recent research work related to the AS-TRM architecture includes the work of Heng Kuang [6]. He also introduced the reliability self-assessment of an Autonomic Reactive Component as part of his master thesis research work [6] [7][9]. The reliability calculation proposed in [6][9] was derived from the theory of Discrete Time Markov's chains and needed as input the exhaustive legal behavior of a group of Autonomic Reactive Objects (ARO). The proposed in [6][7][9] used as input the exhaustive legal behavior calculated manually. The present research work complements the reliability self-assessment calculation method by providing an automated way to generate its input. In references [7] [18], the Autonomic System Specification Language (ASSL), a framework for formally specifying and generating autonomic systems, is used to describe formally the AS-TRM architecture. A case study was conducted by specifying in ASSL the architecture of the Team-Robotics case study [18]; in such case study, autonomic features like self-scheduling and self-monitoring are described.

Manar Abu Talib proposed a formalization of the functional size measurement from the AS-TRM architecture [16] under the COSMIC measurement method developed by the Common Software Measurement International Consortium (COSMIC) [9].

Previous work in the area of formal verification included [8], in which Achuthan points to the use of temporal logic for defining system properties such as safety. Muthiayen [11] specifies formally a set of axioms for defining system properties such as safety states them as theorems and proves those theories by the mean of the PVS theorem prover. However, the approach is iterative and requires a considerable expertise in order to verify the properties of the system. In the research on the algorithms for calculating synchronous composition, the work of Zheng [12] in the automated generation of test cases on reactive systems proposed the SPM algorithm on which this research is based, although the algorithm presented in this research work differs in important aspects. The original synchronizes pairs of objects only, and does not make use of any mechanism for including safety and liveness properties which allow every possible state to be derivable from the system. Nor does it provide a way to handle the time constraints of the resulting composite product. In this research work, those issues are handled by the SPM algorithm presented in Chapter 4 by allowing a complete subsystem to be calculated at the same time. In addition, the algorithm includes the use of safety and liveness system policies as input, thereby providing an alternative to model checking and ensuring correctness by construction. Finally, the interaction of the autonomic reactive objects within the component in terms of time constraints is also properly specified in the final product composition of the autonomic reactive objects.

Our approach differs from the existing work in that the model of the exhaustive behavior of a set of synchronously communicating autonomic reactive along with the time constraints complying with the system policies, is built automatically. On the other side, the correctness of the resulting model is guaranteed by construction because the system policies and timing constraints are built-in and do not require any additional verification.

Chapter 2 will explain in more detail the TROM formalism and its development framework TROMLAB.

Chapter 3. Background

This chapter will explain in more detail the TROM formalism and its development framework TROMLAB. Next, an explanation on how TROM is extended with autonomic capabilities to form AS-TRM is shown.

3.1 Timed Reactive Object Model (TROM) formalism

The Timed Reactive Object Model (TROM) conceived at Concordia University by [4] is a formalism used to model real time reactive systems. It is based on Object Oriented principles that enable it to have features like modularity, reuse, encapsulation and hierarchical decomposition using inheritance. At its structure, TROM is formed by three tiers which work independently to allow different levels of abstraction in the specification as shown in Fig. 1.

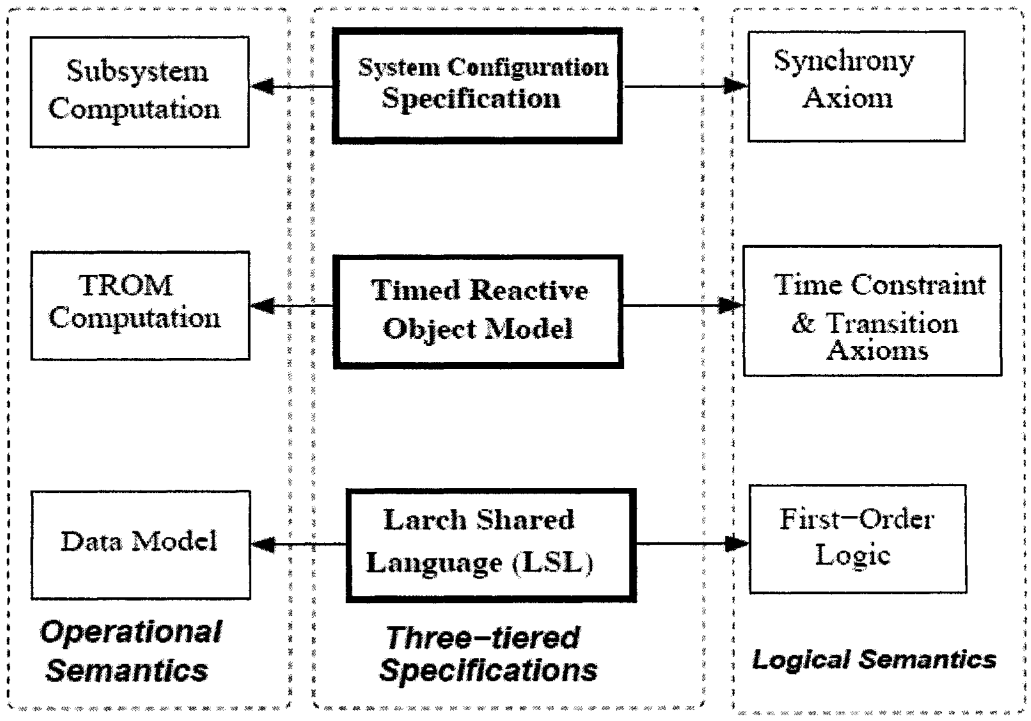

Subsystem Computation
System Configuration Specification
Synchrony Axiom
TROM Computation
Timed Reactive Object Model
Time Constraint & Transition Axioms
Data Model
Larch Shared Language (LSL)
First-Order Logic
Operational Semantics
Three-tiered Specifications
Logical Semantics

Figure 1. TROM architecture; 3 tiers view [4]

The first tier represents the abstract data types expressed by Larch Shared Language (LSL) specification [4] traits at the bottom level. In the middle-tier specification, reactive objects are specified as TROM classes. The top tier specification formalizes collaboration between objects where each object is a TROM object.

3.2 First Tier – Larch Formalism

The Larch Shared Language [5] is used to specify data abstraction meant to be used by the second tier of the TROM model to model the attributes of the generic reactive classes. An example of a Larch LSL is shown next:

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

3.3 Second Tier – TROM Formalism

This tier models the TROM objects or *Generic Reactive Classes (GRC).* A GRC is a hierarchical finite state machine augmented with ports, attributes, logical assertions on the attributes and time constraints. A GRC has a single thread of control and the communication between GRCs is made through message passing also known as *rendezvous*. An interaction represents the act of message passing between the GRC and its environment. A port type is the specification of the messages used in the interaction taking place at a specific GRC with its environment. A state is an abstraction that denotes system information or environmental information at a given moment; a GRC has a unique initial state. An event denotes the GRC's instantaneous activity which can be of three types: incoming, outgoing and internal. The attributes of a GRC can be abstract data types imported from the first tier or port types. The following presents a formal definition of a TROM as an 8-tuple $(P, \mathcal{E}, \Theta, X, \mathcal{L}, \Phi, \Lambda, \Upsilon, R)$ where:

- P with a finite set of ports associated with each port-type, and the null-type P_0 whose only port is the null port $_0$;

- $\mathcal{E}$ is a finite set of events and includes the silent-event tick; the set $\mathcal{E}$ - {tick} is divided into two disjoint subsets:

    - $\mathcal{E}_{ext}$ represents the set of external events, and
    - $\mathcal{E}_{int}$ represents the set of internal events
    - The internal events are associated with the null port P_0
    - Each external event is associated with a unique port-type $p : P - \{Po\}$

- Θ is a finite set of states where $\theta_0 : \Theta$, is the initial state; there is no final state (the behavior of a TROM is assumed to be infinite in time)

- X is a finite set of typed attributes:

  - abstract data types
  - port reference types

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

- Φ is a function-vector (Φ_s, Φ_{at}) where Φ_s associates with each state θ a set of sub states and Φ_{at} associates with each state θ a set of attributes.

- Λ is a finite set of transition specifications; a transition specification λ is a three-tuple: $< \langle\theta, \theta'\rangle\ ;\ e(\psi_{port});\ \psi_{en}\ \Rightarrow\ \psi_{post} >$; where:

  - θ, θ': Θ are the source and destination states of the transition;
  - an event e labels the transition; ψ_{port} (optional) is an assertion on the identifier of the port at which an interaction associated with the transition can occur.
  - $\psi_{en}\ \Rightarrow\ \psi_{post}$, where ψ_{en} is the enabling condition and ψ_{post} is the post-condition of the transition.
  - ψ_{en} is an assertion on the attributes specifying the condition under which the transition is enabled.
  - ψ_{post} is an assertion on the attributes specifying the data computation associated with the transition.

- Υ is a finite set of time-constraints. A time constraint $\nu_i \in \Upsilon$ is a tuple $(\lambda_i, e'_i, [l, u], \Theta_i)$ where:
    - λ_i is a transition specification,
    - e'_i is the constrained event which can be of type internal or external but not input,
    - $[l, u]$ defines the minimum and maximum response times
    - $\Theta_i : \Theta$ is the set of states wherein the timing constraint will be ignored (optional).

An example showing a GRC using the TROM specification language is shown next:

```
Class Pump [@P]
Events: OpenPump?@P, ClosePump?@P, open
States: *closed, toopen, opened
Attributes:
Traits:
Attribute-Function: closed -> {};toopen -> {};
opened -> {};
Parameter-Specifications:

Transition-Specifications:
     R1: <closed,toopen>; OpenPump[](true); true => true;
     R2: <closed,closed>; ClosePump[](true); true => true;
     R3: <toopen,opened>; open[](true); true => true;
     R4: <opened,closed>; ClosePump[](true); true => true;
     R5: <opened,opened>; OpenPump[](true); true => true;
Time-Constraints:
     TCvar1: R1, open, (0, 5), {};
end
```

To specify a subsystem configuration of GRCs we use the composite class specification. This is intended to reduce complexity of the system and to promote modularity at the subsystem level. A Composite Class is a macro-architecture that may include micro-architectures (TROM classes) as well as other macro-architectures [4]. The composition rule that determines the configuration of the components in a macro-architecture is based on port-type compatibility [4]. A composite object can have multiple threads of control. By gluing compatible ports of GRCs a composite class can be formed. The events associated to those glued external ports become internals inside the specification of the composite class. The composite class specification is shown next:

```
CompositeClass<identifier>[<listofprot - types>]
    Incarnations:
    Connectors:
end
```

The specification includes an identifier part to specify the name of the class. An incarnation of a class is the class specification in which the port-type parameters may be renamed. The incarnations section defines a set of incarnations which participates in the composition of the class. The connectors section lists the connectors that glue the compatible ports.

3.4 Third Tier – System Configuration Specification

The System Configuration Specification is used to define system or subsystems specifications by the composition of smaller subsystems or GRCs. Each subsystem is formed by the collaboration of the objects instantiated from the second tier. The template for forming system configuration is shown next:

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

The Subsystem word at the beginning of the specification is followed by its name. Next, there is the Include section to import other subsystems; the instantiate clause to define GRCs; and the configure class defines a configuration obtained by the collaboration of GRCs specified in the Instantiate clause and other subsystems specified in the include section.

3.5 TROMLAB

TROMLAB is a framework created at Concordia University based on the TROM formalism. The following is the current list of components developed and implemented for TROMLAB:

- **Rose-GRC Translator -** which automatically maps the graphic UML model to the formal specification;
- **Interpreter -** which parses, syntactically checks a specification and constructs an internal representation;
- **Simulator -** which animates a subsystem based on the internal representation, and enables a systematic validation of the specified system;

- **Browser for Reuse** - which is an interface to a library, to help users navigate, query and access various system components for reuse during system development;
- **Graphical User Interface** - which is a visual modeling and interaction facility for a developer using the TROMLAB environment;
- **Reasoning System** - which provides a means of debugging the system during animation by facilitating interactive queries of hypothetical nature on system behavior.
- **Verification Assistant** - which is an automated tool that enables mechanized axiom extraction from real-time reactive systems.
- **Validation Tool** - an automated tool that enables mechanized validation of safety properties based on PVS.

The architecture of TROMLAB is shown in fig. 2.

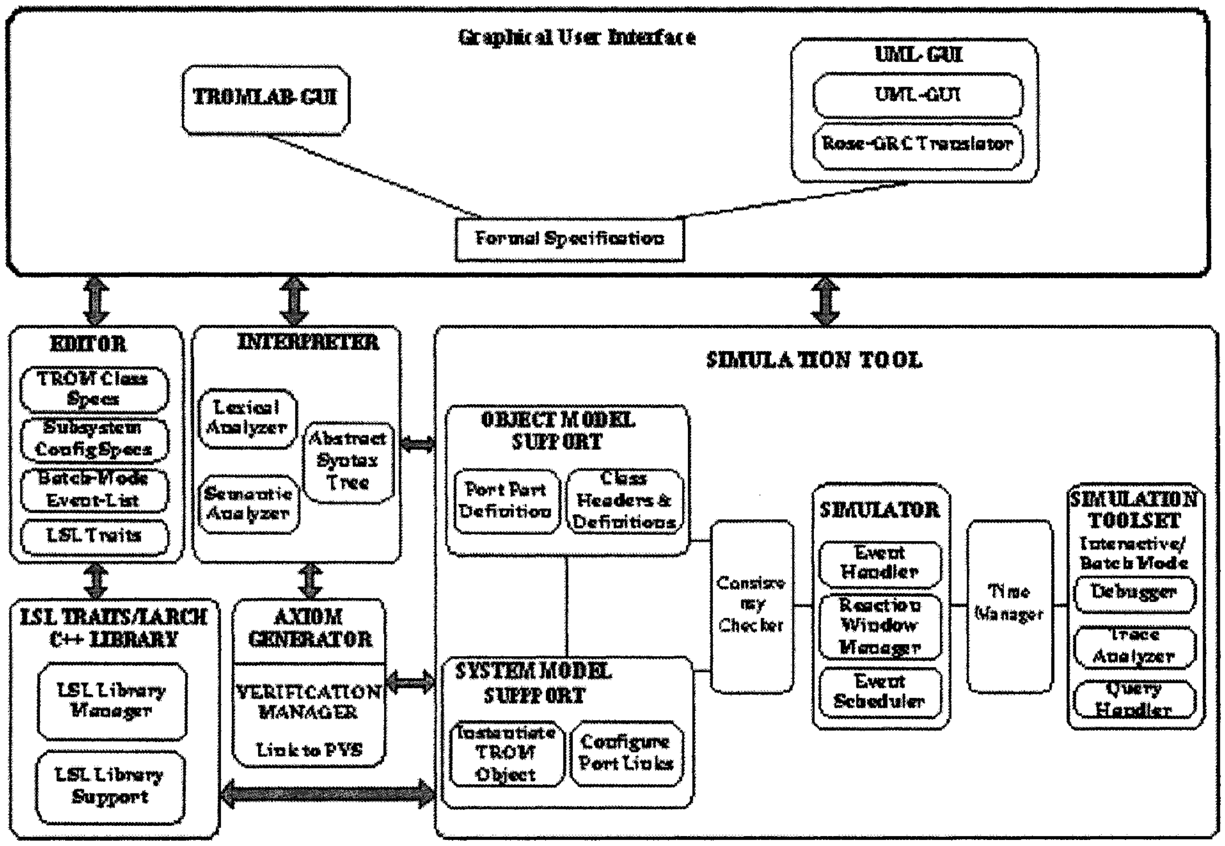


Figure 2. TROMLAB architecture

Next, the AS-TRM is described as a hybrid of both: Autonomic Systems and Real-Time Reactive Systems.

Chapter 4. Autonomic Systems Timed Reactive Model (AS-TRM)

The Autonomic Systems Timed Reactive Model (AS-TRM) is an extension of the TROM formalism to include capabilities of Autonomic Systems [6]. It is a framework for autonomic distributed real-time reactive systems which leverages their modeling, development, integration, maintenance, and continuous monitoring of their reliability [6]. This is shown in figure 3.

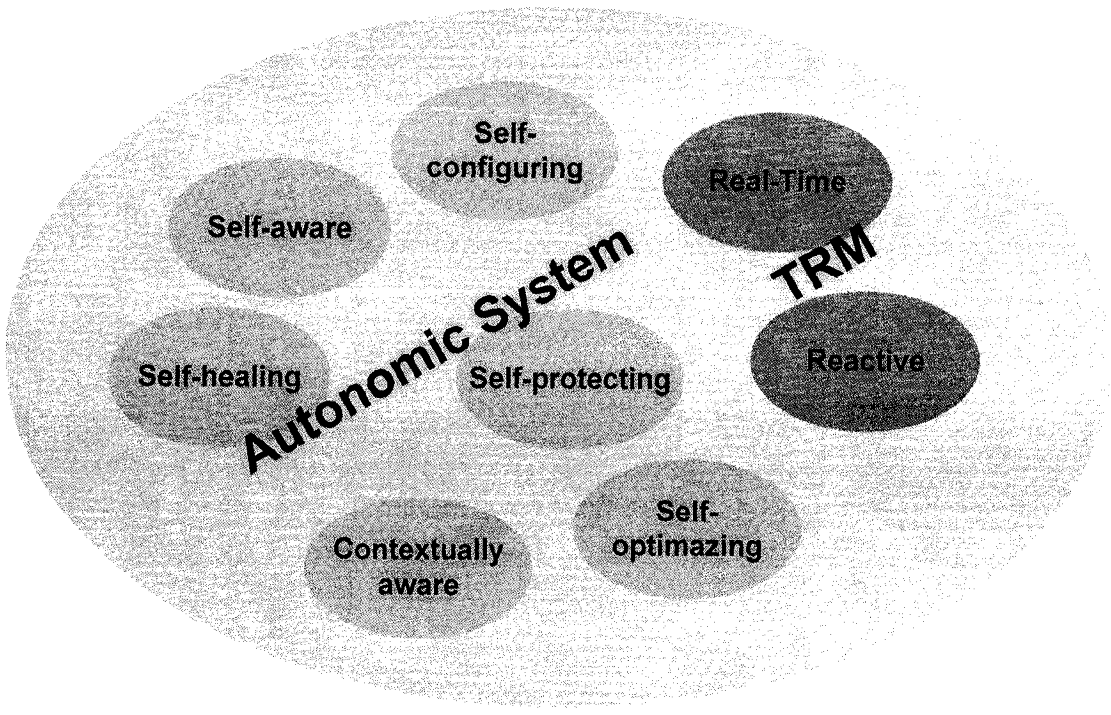


Figure 3. Concept of AS-TRM

The AS-TRM tiers extending the TROM formalism as shown in figure 4.

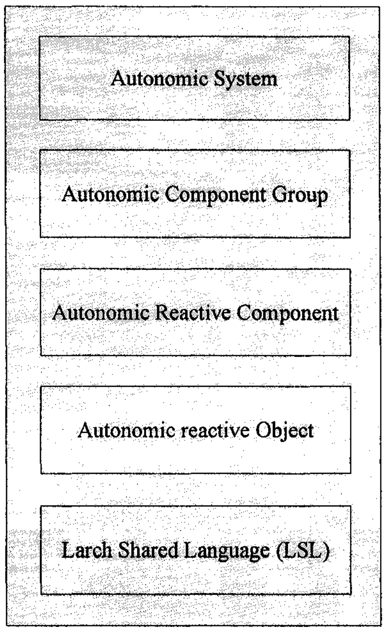

Autonomic System
Autonomic Component Group
Autonomic Reactive Component
Autonomic reactive Object
Larch Shared Language (LSL)

Figure 4. AS-TRM architecture; 5 tiers view

These tiers are explained as following. The Autonomic Reactive Component (ARC) in AS-TRM substitutes the TROM top tier System Configuration, and the Autonomic Reactive Object (ARO) tier substitutes TROM object tier (see Figure 1). In AS-TRM, there are two additional tiers, namely:

- A group of synchronously interacting ARCs: AS-TRM Component Group (ACG);

- A collection of asynchronously interacting ACGs: AS-TRM System (AS).

4.1 Autonomic Reactive Object (ARO)

The Autonomic Reactive Object (ARO) specified in AS-TRM is an extension of the Generic Reactive Component (GRC) expressed in TROM. Like the GRC, the ARO is an extended state machine augmented with ports, attributes, logical assertions on the attributes, and time constraints [4]. Formally, the reactive autonomic object ARO is modeled as a 9-tuple $(P, \mathcal{E}, \Theta, X, \mathcal{L}, \Phi, \Lambda, \Upsilon, R)$ where:

- P is a finite set of port-types; a distinguished port-type is the null-type P_0 whose only port is the null port $_0$; the communication between the AROs is realized through ports from which the events are sent/received by a ARO.
- $\mathcal{E}$ is a finite set of events and includes the silent-event tick; the set $\mathcal{E}$ - {tick} is divided into two disjoint subsets:
    - $\mathcal{E}_{ext}$ represents the set of external events, and
    - $\mathcal{E}_{int}$ represents the set of internal events
    - The internal events are associated with the null port P_0
    - Each external event is associated with a unique port-type $p : P - \{Po\}$
- Θ is a finite set of states where:
    - $\theta_0 : \Theta$, is the initial state; there is no final state (the behavior of an ARO is assumed to be infinite in time)
    - $\theta_p : \Theta$ is a designated state for modeling the planning of the reactive tasks based on the available resources;
- X is a finite set of typed attributes:
    - abstract data types
    - port reference types

- $\mathcal{L}$ is a finite set of LSL traits for the abstract data type used in X
- Φ is a function-vector $(\Phi_s, \Phi_{at}, \Phi_r)$ which Φ_s associates with each state θ a set of sub states, Φ_{at} associates with each state θ a set of attributes and Φ_r associates θ_p with the set R.
- Λ is a finite set of transition specifications; a transition specification λ is a three-tuple: $< \langle\theta, \theta'\rangle ; e(\psi_{port}); \psi_{en} \Rightarrow \psi_{post} >$; where:
  - θ, θ': Θ are the source and destination states of the transition;
  - an event e labels the transition; ψ_{port} (optional) is an assertion on the identifier of the port at which an interaction associated with the transition can occur.
  - $\psi_{en} \Rightarrow \psi_{post}$, where ψ_{en} is the enabling condition and ψ_{post} is the post-condition of the transition.
  - ψ_{en} is an assertion on the attributes specifying the condition under which the transition is enabled.
  - ψ_{post} is an assertion on the attributes specifying the data computation associated with the transition
- Υ is a finite set of time-constraints. A time constraint $\nu_i \in \Upsilon$ is a tuple $(\lambda_i, e'_i, [l, u], \Theta_i)$ where:
  - λ_i is a transition specification,
  - e'_i is the constrained event which can be of type internal or external but not input,
  - $[l, u]$ defines the minimum and maximum response times

    - Θ_i : Θ is the set of states wherein the timing constraint will be ignored (optional).

- *R is a* set of resources available locally for the object to support its functionality. The main differences between TROM and ARO specifications are listed below:

- One more set is included in the extended finite state machine specification of an ARC, namely, a set *R* which models the set of resources available locally for the object to support its functionality.

- Set Θ in ARO includes a designated state θ_p : Θ for modeling the planning of the reactive tasks based on the available resources, which is to model the autonomic behavior of the object;
- The function-vector (Φ_s, Φ_{at}, Φ_r) in ARO is augmented with one more component where Φ_r associates θ_p with the set *R*.

4.2 Autonomic Reactive Component (ARC)

A reactive component in AS-TRM is a collaboration of reactive autonomic objects, where one of the objects is designated as an ARC leader. A reactive component is specified by composing reactive autonomic objects. The specification consists of the following sections: *Members, Configure* and *Leader*; the template is provided below:

```
RC <name>
        Members: <list of reactive autonomic objects>
        Configure: <list of pairs of synchronously communicating objects>
        Leader: <name of the reactive autonomic object designated as a RC
        leader>
End_RC
```

The environmental objects synchronized with the system are modeled as reactive autonomic objects and are included in the ARC.

4.3 Autonomic Components Group (ACG)

The AS-TRM Autonomic Component Group is a set of synchronously communicating ARCs cooperating in a fulfillment of a group task. Each ACG can independently schedule and accomplish a complete real-time reactive task. The self-scheduling and monitoring behavior at the ACG tier level is realized by the ACG's Autonomic Group Manager (AGM). Group configuration specification provides the specification for a group of reactive components by composing them. The template for the ACG is given next:

```
ACG <name>
        Members: <list of reactive autonomic reactive components>
        Configure: <list of pairs of synchronously communicating objects>
        Manager: <name of the ACG manager>
End_ACG
```

4.4 The Global Manager

The AS consists of a set of asynchronously communicating ACGs. The self-monitoring behavior and the asynchronous interaction between the AS and the ACGs is realized by the Global Manager (GM). The responsibilities of the GM include the continuous distribution and monitoring of the AS tasks, and self-healing in occurrence of task failure.

4.5 Characteristics of AS-TRM

The AS-TRM extends the TROM formal model by including the specifications for a time-reactive object (ARO), an autonomic reactive component which consists of a set of synchronously interacting AROs (ARC), and an autonomic system (AS) consisting of asynchronously communicating ACGs.

A diagram illustrating the interaction between the elements in the AS-TRM architecture is shown in figure 5:

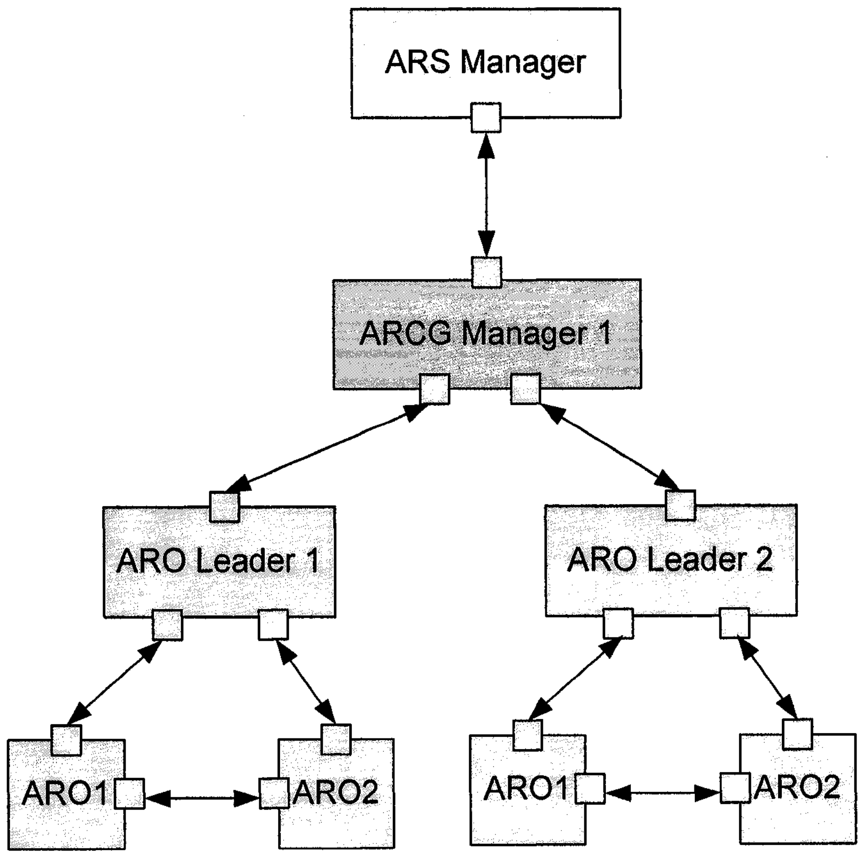


Figure 5. Interaction between elements of AS-TRM architecture

Following there is a summary of the autonomic characteristics of the AS-TRM architecture in addition to the real-time and reactive characteristics inherited from the TROM formalism:

– The AS-TRM is self-managed: it can monitor its components (internal knowledge) and its environment (external knowledge) by checking its status from them, so that it can adapt to changes, known or unexpected, that may occur, which may be the following;

– The AS-TRM is distributed: the components within it can collaborate to complete a common real-time reactive task in a distributive fashion;

– The AS-TRM is proactive: it can initiate changes to the system;

– The AS-TRM is evolving: a) the policies of each AS can be changed during runtime according to changes in the requirements; b) the composition rules of the ARCs within the corresponding peer group can be changed during runtime; c) the synchronization axioms among the AROs and ARCs within the corresponding peer group can be changed during runtime.

Chapter 5 describes the algorithm that generates the synchronous product machine (SPM) of an Autonomic Reactive Component (ARC) containing several Autonomic Reactive Objects (ARO). Also, an algorithm for calculating the minimum and maximum delay time is presented.

Chapter 5. Synchronous Product Machine Algorithm Description

The Synchronous Product Machine (SPM) is a technique for creating complex states from two or more state machines. This complex state represents the exhaustive behavior of the group of the involved state machines. In this chapter, an algorithm to calculate the SPM of a group of Autonomic Reactive Objects is presented.

5.1 Informal Description

The SPM algorithm's goal is to construct a directed-graph data structure representing the SPM of a group of AROs contained in an ARC; the SPM transitions are abstracted as edges and the SPM states as vertices. The synchronous product machine represents all the states and transitions that results of combining two or more machines to produce a complex one. For example, the synchronous product machine (SPM) of two simple state machines S1 and S2 (see Fig.6) is depicted graphically in Fig. 7, where each SPM state is a combination of states of S1 and S2, and the SPM transitions modeling the synchronous communication between S1 and S2 are triggered by the shared events e and f.

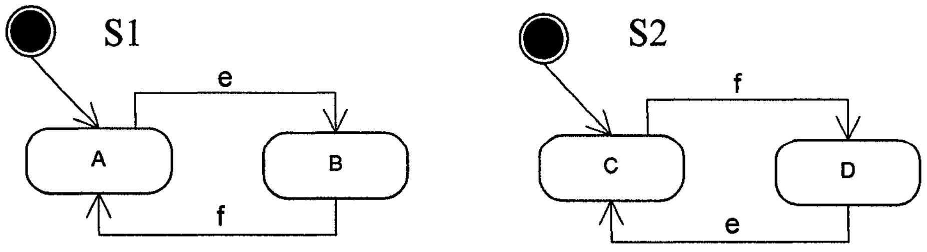


Figure 6. Simple state machines S1 and S2

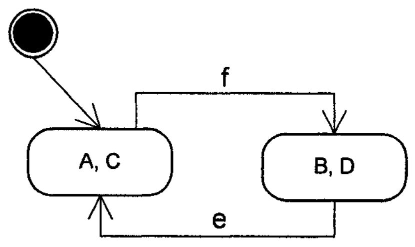


Figure 7. SPM of state machines S1 and S2

The algorithm constructs each new SPM state in a similar way as a breadth-first algorithm [14] traverses the vertices in a graph. It starts by examining the initial SPM state and then generates the SPM states and SPM transitions derivable from that initial SPM state. The set of newly produced SPM states is used to analyze each of their new SPM states and to produce another set of new SPM states. This process stops when no new SPM states can be produced. If a SPM state that does not produce any new SPM transitions is reached then it is marked as erroneous and the algorithm continues analyzing the remaining SPM states. If the algorithm stops with no erroneous SPM states, then this means that the ARC being analyzed according to its AROs synchronous communication specification and the system's policies is valid. Otherwise, the system is erroneous at some point of the ACG design.

The AROs modeled in AS-TRM are extended state machines that communicate synchronously with each other through **external** events, while **internal** events are used within the context of a single ARO. This means that a SPM transition may be generated based on i) a single ARO transition with an internal event, or ii) two (or more) ARO transitions sharing an external event. The generation of the transitions is restricted by a set of policies that implement the safety and liveness properties of the reactive system. This policies are modeled as first-order logical expressions where a set of predicates are related using the logical operators *and*, *or* and *negation*. So, the algorithm does not calculate exhaustively all the **possible SPM states** (which are represented as all possible combinations of the ARO simple states) but rather calculates the valid SPM states complying with the set of policies.

The specifications of the timing constraints for a resulting SPM (see section 5.3.1) are handled by the algorithm in a two-stage process. In the first stage, a clock variable TCVar# is generated for tracing the new SPM time constraint, and in the second stage it gathers the information related to the SPM transitions that are constrained by the TCVar# clock. The correctness of the timing requirements in terms of their conformance to the maximum response delay is checked on the SPM, as described in Section 5.5.6.

5.2 Input Specification

Following the specification of the input to the SPM is given, which consists of a set of AROs and an autonomic reactive component (ARC) configuration specification, along with a set of safety and liveness policies. The ARO specification has already been introduced in chapter 3.

5.2.1 Autonomic Reactive Component Configuration Specification

The template for specifying the ARC configuration specification is shown in section 4.9 of the previous chapter. For the calculation of the SPM algorithm we only use the information regarding the identification of the synchronous communication between AROs, which is specified as a collection of pairs $S = \{\langle O_{a_1}, O_{b_1}\rangle, \dots, \langle O_{a_n}, O_{b_n}\rangle\}$, where two AROs, O_a and O_b , are synchronously interacting *iff* there exists a pair in S such that $\langle O_a , O_b \rangle \vee \langle O_b , O_a \rangle$.

5.2.2 Specification of System Policies in AS-TRM

The behavior of an ARC in AS-TRM is modeled by a set of *safety* and *liveness* policies which are specified at the ACG layer. A mixed state and event-based specification of safety and liveness policies in first order logic is used. A liveness policy specifies that, given a determined state of the system, some specific event has to eventually occur in the ACG. On the other hand, a safety policy specifies that, given a certain configuration of states, a given event has to be prevented from happening. The definitions of the predicates used in defining the timing properties are introduced below:

I. The predicate $object\_at(O_i, \theta)$[15] is used to specify a specific ARO O_i in its state θ or $object\_at(O, \theta)$ to specify AROs of the same type as O in their state θ,

II. The predicate $object\_of(O_i, \theta)$ is introduced to specify that all but one object of type O to is in state θ,

III. The predicate $vector\_at(O, \theta_1, \dots, \theta_n)$ is used to define a set of states of a vector of reactive autonomic objects of the same type O; and

IV. The predicate $occur(O_i, e, \lambda)$ [15] is used to specify an action to take place in the form of an event e in the object O_i; optionally, we may specify λ, the exact transition that triggers event e. For instance, a safety property $object\_at(O_k , \theta_l) \Rightarrow \neg\, occur(O_i, e, \lambda)$ means that for every state $(\theta_1 , \theta_2 , \dots, \theta_n)$ in Θ if the substate corresponding to an object O_k is θ_l then the object O_i cannot make use of the event e and/or the transition λ. On the other hand, a liveness property $object\_at(O_k, \theta_l) \Rightarrow occur(O_i, e, \lambda)$ specifies exactly the same situation for a given SPM state but in this case the predicate *occur()* imperatively prompts the use of the event e and/or transition λ for the analysis of the SPM state.

The safety and liveness properties are specified within the ACG using the following template:

```
ACG_Policies <name>
        Safety: <list of safety properties>
        Liveness: <list of liveness properties>
End_ACG_Policies
```

An example showing one safety policy and one liveness policy is given bellow:

```
ACG_Policies  MyACG_policies
        Safety:
                object_at (Train1, Exit) AND object_at (Controller1,
                monitoring) -> occur (Gate1, Up)
        Liveness:
                vector_at (Train, Idle) AND object_at (Gate1, Open) -> NOT
                occur (Train1, In)
End_ACG_Policies
```

For the Safety policy in the example above the premise part is:

object_at (Train1, Exit) AND *object_at* (Controller1, monitoring)

We can see that the premise has two predicates. The conclusion part is the following:

occur (Gate1, Up)

5.3 Output Specification

The output of the SPM algorithm is represented by the synchronous product machine composition of the ARC being specified. Informally, each state in the SPM is a combination of states, one state from each AROs belonging to the ARC; the transitions, triggering events and timing constraints are defined according to the SPM notation and the SPM generation method, as described below.

5.3.1 SPM Operation Specification in AS-TRM

Mathematically, the SPM operation is defined by the symbol ⊗ [4]. An SPM of n AROs is denoted by $O^{syn} = \{O_1 \otimes O_2 \otimes \dots \otimes O_n\}$ where O^{syn} is a 9-tuple of the form $(P^{syn}, \mathcal{E}^{syn}, \Theta^{syn}, X^{syn}, \mathcal{L}^{syn}, \Phi^{syn}, \Lambda^{syn}, \Upsilon^{syn}, R^{syn})$ such that:

- P^{syn} is a set of port-types allowing for a synchronous communication between the AROs
- $\mathcal{E}^{syn}$ is a union of all $\mathcal{E}_i$ where $i\colon [1..n]$
- Θ^{syn} is a finite set of reachable and valid SPM states; each SPM state is a vector of n states $(\theta_1, \theta_2, \dots, \theta_n)$ where $\theta_i\colon \Theta_i$. A valid SPM state is a state generated by the SPM algorithm for which the ACG policies hold. There is only one initial state notated as $\theta_0^{syn} = (\theta_1^{syn}, \dots, \theta_n^{syn})$, and no final states.
- X^{syn} is a union of the finite sets $X_1^{syn}, \dots, X_n^{syn}$
- $\mathcal{L}^{syn}$ is a union of the finite sets of LSL traits for the ADT used in the AROs
- Φ^{syn} is a function-vector $(\Phi_s^{syn}, \Phi_{at}^{syn}, \Phi_r^{syn})$ which Φ_s^{syn}associates with each SPM state θ^{syn} a set of sub states, Φ_{at}^{syn} associates with each SPM state θ^{syn} the

union of the set of attributes a set of attributes $\Phi_{at\,1}(\theta_1^{syn}), \dots, \Phi_{at\,n}(\theta_n^{syn})$; and Φ_r^{syn} associates each SPM state θ^{syn} with a subset of R^{syn}.

- Λ^{syn} is a finite set of transition specifications, where a transition specification λ^{syn} is a three-tuple: $< \langle\theta^{syn}, \theta'^{syn}\rangle\ ;\ e;\ \psi_{en}^{syn} \Rightarrow \psi_{post}^{syn} >$ described below:
  - $\theta^{syn}, \theta'^{syn}$: Θ^{syn} are the source and destination states of the transition.
  - an event e labels the transition
  - There are two cases to be considered when specifying $\psi_{en}^{syn} \Rightarrow \psi_{post}^{syn}$, namely:
    - e is an internal event for a single substate θ_i^{syn}. In this case $\psi_{en}^{syn} = \psi_{en\,i}$ and $\psi_{post} = \psi_{post\,i}^{syn}$
    - e is a shared (common) external event for the reactive objects $O_1, O_j, O_k, \dots$ which triggers a change of state in the θ'^{syn} substates $\theta_1, \theta_j, \theta_k, \dots$ In this case, ψ_{en}^{syn} is a disjunction of the enabling conditions $\psi_{en\,i}, \psi_{en\,j}, \psi_{en\,k}, \dots$ and ψ_{post}^{syn} is the conjunction of the post-conditions of the transitions $\psi_{post\,i}, \psi_{post\,j}, \psi_{post\,k}, \dots$
- Υ^{syn} is a finite set of time-constraints. A time constraint $\nu_i^{syn} \in \Upsilon^{syn}$ is a tuple $(\lambda'^{syn}_i, \mu'^{syn}_i, [l^{syn}, u^{syn}], \Theta_i^{syn})$ where:
  - λ'^{syn}_i is a transition specification,
  - μ'^{syn}_i is a set of 2-tuples of the form $(e'^{syn}_i, \lambda'^{syn}_i)$ where $e'^{syn}_i \in \lambda'^{syn}_i$
    - e'^{syn}_i is the constrained SPM event which can be of type internal or external in the definition of the corresponding reactive object(s) $O_1, O_j, O_k, \dots$

    - λ'^{syn}_i is the SPM transition specification that issues e'^{syn}_i
  - $[l^{syn}, u^{syn}]$ define the minimum and maximum response times. There are two cases to be considered when specifying $[l^{syn}, u^{syn}]$, namely:
    - e is an internal event for a single substate θ_i^{syn} of an object O_m. In this case $l^{syn} = l_i$ and $u^{syn} = u_i$, where $[l_i, u_i]$ is the range defined for this transition in O_m.
    - e is a shared (common) external event for the reactive objects O_i, O_j, O_k, … which triggers a change of state in the θ^{syn}' substates θ_i^{syn}, θ_j^{syn}, θ_k^{syn}, … In this case $[l^{syn}, u^{syn}]$ is set to be the most restrictive range $[l_m, u_m]$ for the transitions originating in the states θ_i^{syn}, θ_j^{syn}, θ_k^{syn} … and due to e; we choose only those of the reactive objects O_i, O_j, O_k, … where e is defined as an output external event (the input external events cannot be restricted due to the reactive behavior of the object).
  - $\Theta_i^{syn}: \Theta$ is the set of states wherein the timing constraint will be ignored (optional). The set Θ_i^{syn} is generated by the intersection of the those substates from the set $\theta_1, \theta_2, \ldots, \theta_n$ for which the timing constraints are ignored in their corresponding objects O_i, O_j, O_k, …

- R^{syn} is a set of resources available in the ARO; it is defined as a union of all $R_i: [i \ldots n]$.

The timing constraints are built into the model as guards on the transitions; the local clocks are employed for specifying the timing constraints.

5.4 SPM Pseudocode – Main Procedure

This section describes the SPM at its top level. Also, an activity diagram shown in figure 8 depicts the SPM workflow.

Input

```
Formal specification of AROs, ARC policies and ARC
configuration specification
```

Output

```
SPM specification
```

Variables

```
Let C be a set of newly generated SPM states
Let G  be a set of all reachable and valid SPM states
Let T  be a set containing the SPM transitions produced
Let Temp be the SPM state currently being analyzed
```

Calculation

```
1      Start at the initial SPM state $\theta_0^{syn}$ and add it to C
       and G (note: the initial state is both reachable and
       valid)
2      While C is non-empty do:
2.1        Choose an unmarked SPM state from C and assign it
           to Temp
2.2        Select applicable policy to Temp
2.3        If policy of type liveness applies then generate
           candidate transitions from this policy using sub
           procedure described in section 5.5.4
2.4        Else generate candidate transitions from Temp
           using sub procedure described in section 5.5.4
2.5        Validate all candidate transitions against safety
           policy using sub procedure described in section
           5.5.2
```

2.6 If no new valid transitions are generated then mark ***Temp*** as erroneous and continue to analyze next SPM state in ***C***. Go to ***Step 2.1***

2.8 Use validated candidate transitions to construct new SPM transitions and SPM states using the sub procedure described in section 5.5.1

2.7 Add new SPM states to ***C*** and ***G***

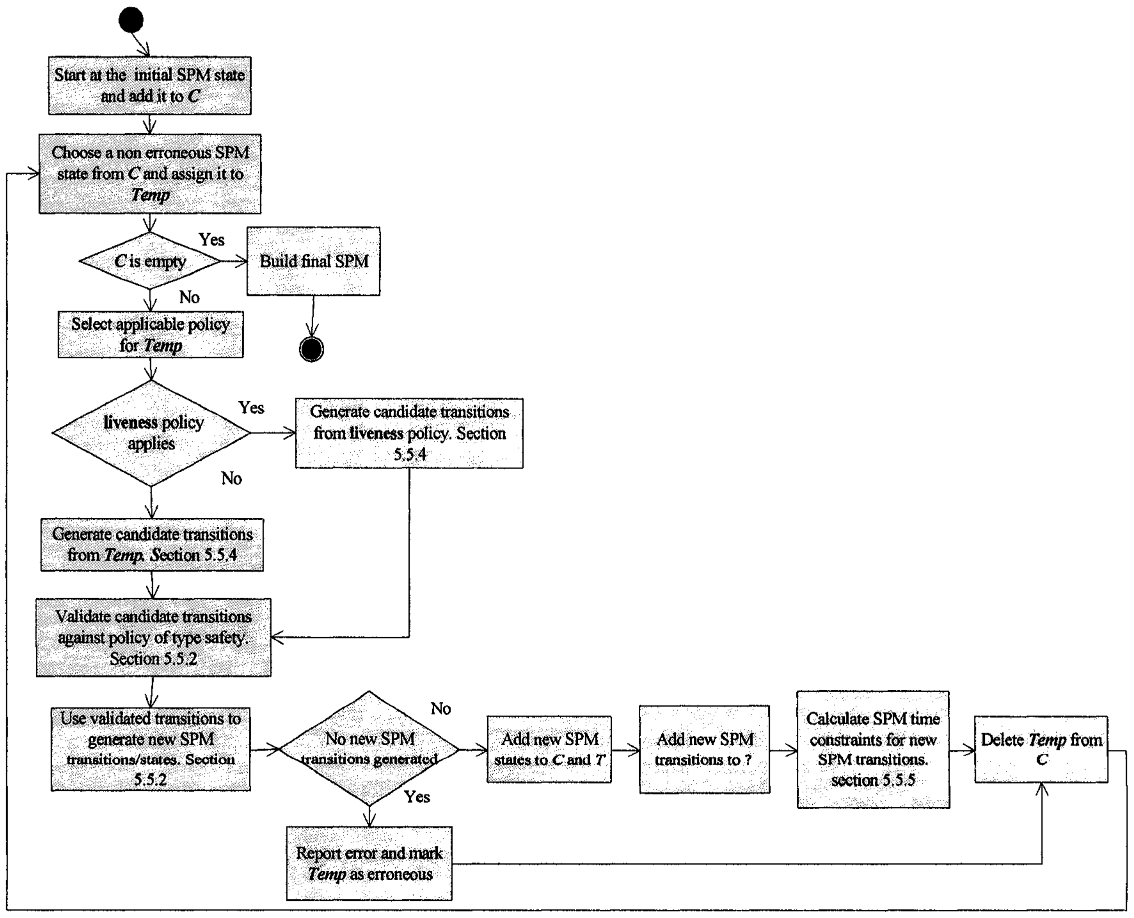
2.8 Add new SPM transitions to ***T***

2.9 Calculate SPM time constraints for newly added SPM transitions using sub procedure described in section 5.5.5

2.10 Delete ***Temp*** from ***C***

End While (*Step 2*)

3 The final SPM is a set that contains the sets of SPM states ***G***, SPM transitions ***T*** and SPM time constraints ***T***



Start at the initial SPM state and add it to C
Choose a non erroneous SPM state from C and assign it to Temp
C is empty
Yes
Build final SPM
No
Select applicable policy for Temp
liveness policy applies
Yes
Generate candidate transitions from liveness policy. Section 5.5.4
No
Generate candidate transitions from Temp. Section 5.5.4
Validate candidate transitions against policy of type safety. Section 5.5.2
Use validated transitions to generate new SPM transitions/states. Section 5.5.2
No new SPM transitions generated
No
Yes
Add new SPM states to C and T
Add new SPM transitions to ?
Calculate SPM time constraints for new SPM transitions. section 5.5.5
Delete Temp from C
Report error and mark Temp as erroneous

Figure 8. Process diagram of SPM algorithm

5.5 SPM Pseudocode – Sub Procedures

In this section, each sub procedure mentioned in the previous section for the algorithm is described in a more detailed view.

5.5.1 SPM Transition and SPM State construction

A SPM state is made by substituting the destination state θ_i' of a candidate transition λ_i into a given SPM state θ_i^{syn}. A SPM transition λ^{syn} can be built based in one or two candidate transitions. If the candidate transition has an event that is internal to one of its ARO, then we build the SPM transition based solely on that transition (see Case 1 below). On the other hand, if two candidate transitions share the same event of type external, then the corresponding AROs are synchronized by that event; the SPM transition is constructed as a merge of the guard conditions (including the timing constraints) and actions of those transitions, and of the initialized clocks (see case 2 below). The timing constraints are expressed using the SPM clock variables.

Case 1. SPM state and transition construction – one candidate transition

```
Input
$\theta_i^{syn}$ = Current SPM state being analyzed
$\lambda_i$ = a candidate transition triggered by  an internal event
Output
New SPM state  $\theta_i^{'syn}$, new SPM transition $\lambda^{syn}$
```

Calculation

1 Create a new SPM state $\theta_i^{syn}{}'$ such that $\theta_i^{syn}{}' = (\theta_1, \dots, \theta_i', \dots, \theta_n)$ where $\theta_i{}'$ is the destination state of transition λ_i

2 Create a new SPM transition λ^{syn} based in λ_i and θ_i^{syn} where $\lambda^{syn} = \langle\langle\theta_i^{syn}, \theta_i^{syn}{}'\rangle; e; \psi_{en}^{syn} \Rightarrow \psi_{post}^{syn}\rangle$ where $\psi_{en}^{syn} = \psi_{en\,i} \wedge \psi_{post} = \psi_{post\,i}^{syn}$

Case 2. SPM state and transition construction – two candidate transitions

Input

θ_i^{syn} = Current SPM state being analyzed

$\lambda_i =$ a candidate transition triggered by an external event e

$\lambda_y =$ a candidate transition triggered by the external event e

Output

New SPM state $\theta_i^{syn}{}'$, new SPM transition λ^{syn}

1 Create a new SPM state $\theta_i^{syn}{}' = (\theta_1, \dots, \theta_i', \dots, \theta_y', \dots, \theta_n)$ where $\theta_i{}'$ is the destination state of the transition λ_i , and $\theta_y{}'$ is the destination state of the transition λ_y.

2 Create a new SPM transition λ^{syn} based in λ_i and λ_y and θ_i^{syn} where $\lambda^{syn} = \langle\langle\theta_i^{syn}, \theta_i^{syn}{}'\rangle; e; \psi_{en}^{syn} \Rightarrow \psi_{post}^{syn}\rangle$ where ψ_{en}^{syn} is a disjunction of the enabling conditions $\psi_{en\,i}$, $\psi_{en\,y}, \dots$ and ψ_{post}^{syn} is the conjunction of the post-conditions of the transitions $\psi_{post\,i}$, $\psi_{post\,y}, \dots$

5.5.2 Evaluation of system policies

This sub procedure calculates if a given SPM state θ_i^{syn} holds for a given policy. This process takes as input a SPM state θ_i^{syn} and returns the policy p_j that holds in θ_i^{syn} or *empty* if no policy holds in θ_i^{syn}. This sub procedure has to deal with the premise part of the logical expression of the policy that is common to all policies (see section

5.2.2). The actual execution of the conclusion of the policy is carried over by the safety check procedure (see section 5.5.3) for safety policies and by Generation of Candidate Transition sub procedure for liveness properties (see section 5.5.4).

Input

A SPM state θ_i^{syn}, a vector of AROs $Q = \langle O_1, \ldots, O_n \rangle$, a set of policies $P = \{p_1, \ldots, p_n\}$

Output

A policy p_i , a set of policies P

Variables

Let p_result be a boolean variable to store the result of the evaluation of a policy p_i

Calculation

```
1         For each policy $p_i$ in P  do
1.1         Set $p\_result = false$
1.2         For each predicate $b_j$ in the premise part of $p_i$  do
1.2.1         If $b_j$ evaluates to true for $\theta_i^{syn}$ then
1.2.1.2         $p\_result = true$; GOTO End for (step 1.2)
1.2.1.3         else $p\_result = false$   // This policy $p_i$ does not
                hold for $\theta_i^{syn}$ and we continue to examine the
                next policy $p_{i+1}$
              End if (step 1.2.1)
            End for (step 1.2)
1.3         If $p\_result = true$ then
1.3.1         $p_i$ holds for $\theta_i^{syn}$ and we return $p_i$ as a result of
              this function
1.3.2         else  no $p_i$ policy in P holds for $\theta_i^{syn}$ and the
              sub procedure returns empty
            End if (step 1.3)
          End for (step 1)
```

5.5.3 Safety Check

A safety property means that some event has to be prevented from happening. The conclusion part of a policy of type safety uses the predicate $\neg\, occur(O_i, e, \lambda)$ to specify which event e we want to prevent from happening (see section 4.2.2). A function $safety\_check\ : \Lambda \to Boolean$ can evaluate if a given candidate transition can be employed in the construction of a SPM transition. In case of an external event, both transitions triggering a change of state in the individual AROs are evaluated by the $safety\_check$ function. Should $safety\_check$ function return *false* for any of those candidate transitions then the SPM candidate transition is discarded.

5.5.4 Generation of Candidate Transitions

This sub procedure's purpose is to generate a new SPM transitions from the applicable individual AROs transitions. The sub procedure analyzes each ARO and selects the applicable ones in the ARO current state's transitions. If the selected transition has a shared external event with another ARO, then the procedure has to find another transition in another ARO that matches the same shared event. For doing that, the ARC configuration specification provides information regarding which AROs are synchronized by external events. The procedure has one exception, that is, when a **liveness** policy applies. In that case, all the candidate transitions will be generated according to the specification included in the conclusion part of that policy.

Input
θ_i^{syn} = Current SPM state being analyzed
p_i = Current policy selected for θ_i^{syn}; see section **4.2.1**

Output

Candidate transition λ_i or candidate transitions λ_y and λ_i at each iteration

Calculation

```
1          For each SPM sub state $\theta_i$ in $\theta_i^{syn}$ do
1.1          If $p_i$ is a policy of type **liveness** then
1.1.1          Get a subset of transitions $\Lambda_i{}'$ from $\Lambda_i$ such
               that each transition $\lambda_i$ in $\Lambda_i{}'$ has a source state
               $\theta_i = \theta_k$ and an event $e_i = e_k$ or a transition $\lambda_i = \lambda_k$
               where $\theta_k, e_k, \lambda_k$ are elements of the conclusion part
               of the policy $p_k$
1.1.2          Else if ($p_i$ is not a policy of type **liveness**) or
               (no policy applies to $\theta_i^{syn}$ ) then
1.1.2.1          Get a set of transitions $\Lambda_i{}'$ from $\Lambda_i$ such that
                 each transition $\lambda_i$ in $\Lambda_i{}'$ has a source state
                 $\theta = \theta_i$
             End if (*step 1.1*)
1.2          For each transition $\lambda_i$ in $\Lambda'$ do
1.2.1          If the event $e$ that triggers the transition $\lambda_i$ is
               of type **internal** then
1.2.1.1          *Perform Safety evaluation, SPM time constraint
                 analysis; SPM transition build and SPM state
                 build procedures for the candidate transition
                 $\lambda_i$*
               End if (*step 1.2.1*)
1.2.2          If the event $e$ which triggers the transition
               $\lambda_i$and the event  is of type **shared** then
1.2.2.1          Get a subset $Q'$ of AROs from $Q$ such that for
                 all the AROs in $Q$ the ARO $O_y$ holds at least one
                 relationship in $R$ such that $\{O_y, O_i\} \vee \{O_i, O_y\}$ and
                 such that $O_y$ contains at least one transition
                 $\lambda_y$ such that its triggering event $e = e'$ where
                 event $e'$ is the triggering event of transition
                 $\lambda_i$ and $e$ is the triggering event of transition
                 $\lambda_y$
1.2.2.2          For each $O_y$ in $Q'$ do
```

```
1.2.2.2.1       Get a subset of transitions $\Lambda''$ from $\Lambda$ such
                that each transition $\lambda_y$ in $\Lambda''$ has a source
                state $\theta_h = \theta_i$ and an event $e_h = e_i$ where $\theta_i$ is
                the source state and $e_i$ is the triggering
                event of transition $\lambda_i$
1.2.2.1.2       For each transition $\lambda_y$ in $\Lambda''$ do
1.2.2.1.2.1       Perform Safety evaluation, SPM time
                  constraint analysis, SPM transition build
                  procedures and SPM state build for
                  candidate transitions $\lambda_y$ and $\lambda_i$
                End for (step 1.2.2.1.2)
              End for (step 1.2.2.2)
            End if (step 1.2.2)
          End for (step 1.2)
        End for (step 1)
```

5.5.5 Time Constraint Calculation

The calculation of the time constraints for a Synchronous Product Machine (SPM) is done partially in a separate sub procedure as well as in the main procedure of the SPM algorithm. The reason for doing this is that for a SPM time constraint v_i^{syn} there can be several constrained events within μ'_i as opposite to a single ARO time constraint which only constrains one event e'_i. This is because at the moment of producing a new SPM transition we do not know which SPM transitions are produced with that time-constrained event/transition e'_i until λ'_i is used in the algorithm process. Therefore, in our implementation, we keep track of λ'_i and every time a new SPM transition λ_j^{syn} is created using λ'_i , we generate a new 2-tuple $(e_j^{syn}, \lambda_j^{syn})$ and we add it to μ'_i. This means that the complete time constraint calculation will be known at the end of the execution of the algorithm. The following sub procedure receives a SPM transition to be analyzed and a set of one or two ARO transitions that collaborate in the construction of the SPM transition received. The objective is to initialize all possible SPM time constraint (clock) variables v_i^{syn} for λ_i^{syn}. The resulting Time Constraint variables will be partially-generated because μ'_i will be empty at this point.

Input

One SPM transition λ_i^{syn}; Set of ARO transitions $T = \{\lambda_1, \ldots, \lambda_n\}$;

Output

Set of newly partially-generated Time Constraint (clock) variables

Calculation

```
1       For each transition $\lambda_i$ in $T$ such that $\lambda_i$ is a
        transition that initializes at least one time
        constraint $v_i$ in $\Upsilon_i$ do
1.1       For each time constraint $v_i$ in $\Upsilon_i$ where $\lambda_i$ is the
          transition initiating $v_i$ do
1.1.1       Create a new SPM time constraint $v_j^{syn}$ where
            $\lambda_j^{syn} = \lambda_i^{syn}$, $\mu'_i = empty$, $[l^{syn}, u^{syn}] = [l_i, u_i]$, $\Theta_i^{syn} = \Theta_j$
1.1.2       Add $v_j^{syn}$ to $\Upsilon^{syn}$
          End for (step 1.1)
        End for (step 1)
```

5.5.6 Min-Max Delay Time Constraint Analysis

This is a static analysis of the time constraint variables generated by the SPM algorithm. Its purpose is to calculate the minimum and maximum time that a given path in the system takes to perform a full period in the SPM. This approach is favoured by [13] as a simple way to calculate time boundaries in a reactive system. The maximum and minimum time of all the paths in the SPM gives the minimum and maximum time of the complete system.

Input

SPM $M = O_1 \otimes \ldots \otimes O_n$

Output

Minimum and Maximum delay time for M denoted by the variables lZ^{syn} and uZ^{syn}

Initialization

Use a depth-first search algorithm [14] to identify and collect all the possible paths in a SPM machine. Let a path z be a sequence of SPM transitions such that $z_i = \langle \lambda_1^{syn}, \ldots, \lambda_n^{syn} \rangle$; also, let Z^{syn} be the set of all paths in a SPM such that $Z^{syn} = \{z_1, \ldots, z_n\}$.

Variables

Let lZ^{syn} and uZ^{syn} be the maximum and minimum time for the current SPM sequence of paths in Z^{syn}

Let *temp_lZ* and *temp_uZ* variables to hold temporary values of lZ^{syn} and uZ^{syn}

Calculation

```
1         $lZ^{syn} = 0, uZ^{syn} = 0, temp\_lZ = 0, temp\_uZ = 0$
2         For each path $z_i$ in $Z^{syn}$ do
2.1         Set $lz^{syn}$ and $uz^{syn}$ be the maximum and minimum time
            for the current path $z_i$
2.2         Set $temp\_lz$ and $temp\_uz$ variables to hold temporal
            values of $lz^{syn}$ and $uz^{syn}$
2.3         $lz^{syn} = 0, uz^{syn} = 0, temp\_lz = 0, temp\_uz = 0$
2.4         For each SPM transition $\lambda_i^{syn}$ in $z_i$ do
2.4.1         Set $v_j^{syn}$ to be a time constraint such that
              $v_j^{syn} \in Y^{syn}$ and $\lambda_i^{syn} = \lambda_j^{syn}$
2.4.2         For each $v_j^{syn}$ in $Y^{syn}$ do
2.4.2.1         $temp\_lz$ is equal to the lowest value for all
                the instances of $l^{syn}$ in $v_j^{syn}$
2.4.2.2         $temp\_uz$ is equal to the highest value for all
                the instances of $u^{syn}$ in $v_j^{syn}$
              End for (step 2.4.2)
2.4.3         If $v_j^{syn}$ is the first occurrence in $z_i$ then
              $lz^{syn} = temp\_lz$ and $uz^{syn} = temp\_uz$
```

```
2.4.4       If   $temp\_lz < lz^{syn}$  then $lz^{syn} = temp\_lz$
2.4.5       else   If  $temp\_uz > uz^{syn}$  then $uz^{syn} = temp\_uz$
          End for (step 2.4)
2.5       If $z_i$ is the first occurrence in $Z^{syn}$ then
          $lZ^{syn} = temp\_lZ$ and $uZ^{syn} = temp\_uZ$
2.6       If $temp\_lZ < lZ^{syn}$ then $lZ^{syn} = temp\_lZ$
2.7       else If $temp\_uZ > uZ^{syn}$  then $uZ^{syn} = temp\_uZ$
        End for (step 2)
```

5.6 Disadvantages of the approach

The SPM algorithm is based in the breadth-first search algorithm [14]. This algorithm is characterized for having a time complexity of $O(\mid V \mid + \mid E \mid)$ [14] where *V* is the number of vertices and *E* the number of edges in the graph structure. In a SPM the vertices are the states and the edges the transitions. The SPM algorithm has different sub procedures that increases the running time compared to a normal breadth-first algorithm that makes its node search without any additional sub procedure. Like a breadth-first algorithm that is exponential in the depth of the graph, the SPM algorithm is impractical for large inputs, specially as a run-time algorithm. However, the aim of the algorithm is to assist on the correctness of the ARC model at design time or before an evolutionary change is authorized and implemented. The depth of the size of the resulting graph in the SPM algorithm is affected by the so called state explosion in the same way as the algorithms used for real-time systems model checking do [13]. This problem arises when the ARC has many AROs that can produce SPM transitions triggered by external events. In such case, the number of resulting SPM states may grow exponentially with the number of calculations [13]. Nevertheless, the increased capacity of processing power of modern computational equipment as well as advances in lighter data structures and implementation techniques [13] makes the proposed approach practical.

5.7 Advantages of the approach

By executing the SPM algorithm, the SPM specification of a group of synchronously communicating AROs is obtained; and therefore, an exhaustive knowledge on the behavior of the ARC. The SPM specification represents all valid behavioural paths of an ARC along with the corresponding states and transitions that a given ARC can manifest during the execution according to the specification of the AROs. This has the following benefits:

5.7.1 Detection of undesired states product of an erroneous design.

Each time the algorithm marks a SPM state that does not produce any transition, it is detecting a flaw in the design of the ARC specification. If an implementation of the algorithm includes a log file of events to track all the calculations that lead to that erroneous SPM state then a designer can know the reasons behind the production of such erroneous SPM state. For example, a safety policy may have held the transitions needed for the erroneous SPM state to produce new SPM transitions; if that is the case, a further analysis of the safety policies may solve the problem.

5.7.2 Verification of the ARC against its specifications.

If an error occurs during the execution of the algorithm it can be because of the following causes:

- An error at the specification of the ARC policies,

- An error at the specification of the ARC configuration specification,
- An error at the specification of the AROs in the ARC,

Also, an error which is more difficult to spot comes when the logic behind the design of the ARC is incorrect. In such case, a designer should have checked that no error of the type shown in the previous list is present before considering modifying the ARC logic design.

5.7.3 Validation of the critical timing requirements

The algorithm that calculates the minimum and maximum delay times for a SPM gives insight on the performance of an ARC. The minimum and maximum delay times asses for the best and worst case scenario that a given ARC can be expected to perform. Also, it can calculate that information of each possible scenario (path) that the ARC can take. This information could be useful in different autonomic applications like:

- Self-estimation of system resources to aid at the planning phase of the autonomic control loop.
- Self-optimization and self-configuration of the system to match delay times that may be imposed as a high-level policy.
- Detection of errors or/and unwanted system behavior if the AC does not perform between the boundaries of the specified maximum and minimum delay time.

5.7.4 Provide with a data repository to be used in autonomic features

The data that the SPM of an ARC produces can be useful for implementation of different autonomic features in AS-TRM. The self-control of the behavioral correctness is one of them because the SPM represent the exhaustive behavior of an ARC; thus, a data repository loaded with that information can detect discrepancies from the actual behavior of an ARC and the expected one in the data repository.

5.7.5 Provide the input for the calculation of the reliability self-assessment in AS-TRM

The self-assessment of the AS-TRM reliability proposed by [8], calculates the reliability of an ARC. Since the input for that calculation is the SPM of the AROs involved in that calculation, this work complements it.

In the next chapter the description of the design of the implementation of the SPM algorithm is presented. Its usage is exemplified with a case study and the results of the experimentation are presented.

Chapter 6. SPM Calculation Tool

In this chapter the description of the design and the implementation of the SPM algorithm are presented. Its usage is exemplified with a case study and the results of the experimentation are presented.

5.1 Design

The implementation of the SPM algorithm was made in accordance of the following architecture:

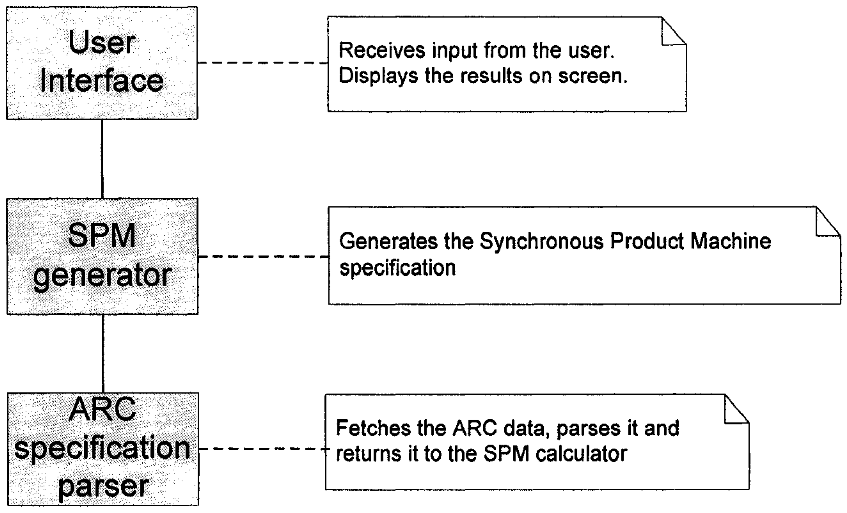

User Interface
Receives input from the user. Displays the results on screen.
SPM generator
Generates the Synchronous Product Machine specification
ARC specification parser
Fetches the ARC data, parses it and returns it to the SPM calculator

Figure 9. Architecture of the implementation for SPM algorithm

User Interface. The user interface has as responsibilities to provide the user with an interface to the system and to display the results on the screen. The user can select between different case study configurations to calculate and decide where to save the resulting output of the calculation.

SPM generator. The SPM generator carries over the whole calculation of the SPM algorithm with all its sub procedures.

ARC Specification Parser. The ARC specification parser fetches the data from text files representing the specification of the ARC system. It also prepares the data structures that the SPM generator will use.

The following collaboration diagram shows the data flow of the overall process:

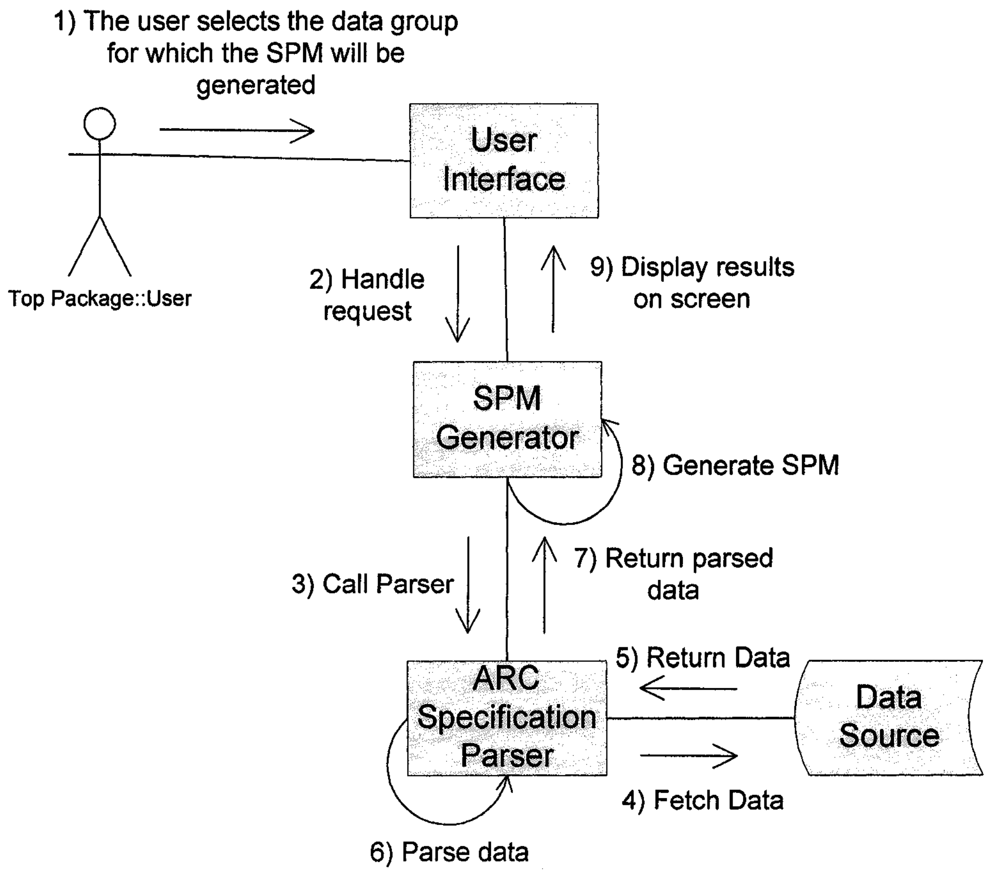


Figure 10. Collaboration diagram for SPM algorithm implementation

6.2 Implementation

AS-TRM [6, 7, 19] derives its ARC functionality and specification by extending the TROM formalism [4] which is based on object-oriented principles. Hence, an object-oriented language is chosen to simulate its behavior. Java version 1.6 as a programming language and NetBeans version 6.10 M10 as a development environment were used to develop the implementation.

The program receives the input from the user as a case study choice. Then, it starts calculating the SPM while providing information of different aspects of the calculation as a log of events. More than only for debugging purposes at development phase, the event log is important in an implementation of an algorithm like this because it provides insight in the case study's specification design. If the specification of a case study is incorrect at some point, the event log will show the succession of events that led to that error. This is shown when the algorithm can not generate an appropriate SPM transition for a given SPM state. In this case, it means that a design error is present in the specification. By examining the event log it is possible to determine if at the moment of analyzing a given SPM state any policy interfered with the use of any potential transition. Also, the logic behind the design of the ARC may be the error's cause. A sample of the event log is presented next showing an erroneous design:

```
Now analyzing SPM_state: [monitor, idle, closed, open, tocross, tocross]

-----------------------

...

... (More event results goes here)

...

----POLICY NUMBER 0 HOLDS----
```

```
Now analyzing SPM_substate: [tocross] in RC: TRAIN1 for SPM_state: [monitor, idle, closed, open, tocross, tocross]

-----CANCELED BECAUSE OF SAFETY POLICY: S1

Evaluating candidate transition: RC: train1 , r4 , [tocross] , [cross] , in , true=>true ,

-----CANCELED BECAUSE OF SAFETY POLICY: S1

Evaluating candidate transition: Now analyzing SPM_substate: [tocross] in RC: TRAIN2 for SPM_state: [monitor, idle,
closed, open, tocross, tocross]

-----CANCELED BECAUSE OF SAFETY POLICY: S1

Evaluating candidate transition: RC: train2 , r4 , [tocross] , [cross] , in , true=>true ,

-----CANCELED BECAUSE OF SAFETY POLICY: S1

--Finished Analysis for SPM_state: [monitor, idle, closed, open, tocross, tocross]

************************ATENTION: SPM STATE ERRONEOUS****************************

[monitor, idle, closed, open, tocross, tocross] Did not produced any new transition. Iteration: 5
```

Taking a look at the event log, which has been shortened due to limitations in space above, it can be deduced that the SPM state: **[monitor, idle, closed, open, tocross, tocross]** did not produced any new SPM transition because safety policy **S1** held the possible candidate transitions for producing any SPM transition. In this case, a further look to the logic of the design of the policies section is indicated towards a successful production of the SPM. This example shows how the algorithm can be useful verifying the specifications of an ARC and help in detecting errors in their design.

In chapter 7, the testing of the algorithm presented in chapter 5 is described. The algorithm is tested with the railroad case study with several possible configurations, and the results of the algorithm performance are reported.

Chapter 7. Railroad Crossing Case Study

In this chapter, the testing of the algorithm described in chapter 3 is described. The algorithm is tested against the railroad case study with several possible configurations and the results are presented.

7.1 Description

The input for the algorithm is given as the formal specifications of the Railroad Crossing case study and is tested with five different configurations. Also, it is presented how the policies that will handle the liveness and safety properties are designed and embedded in the ARC configuration specification. The Railroad Crossing is a long-time used standard case study for testing real-time systems' specification methods The Railroad Crossing case study consists of one or several trains which cross a crossroad with a gate independently and simultaneously using non-overlapping tracks. A controller controls each gate. When a train approaches the gate, it sends a message to the associated controller. Then, the controller commands the gate to close. When the train exits the crossing it sends a message to the controller which instructs the gate to open. The following are the time restriction for the systems to ensure safety:

- A train enters the crossing within an interval of 2 to 4 time units after informing the controller that it is approaching.
- A train informs the controller that it is leaving the crossing within 6 time units of sending the approaching message.
- The controller instructs the gate to close within 1 time unit after receiving the first approaching message and starts monitoring the gate.

- The controller continues to monitor the closed gate when it receives an approaching message from other trains, and as long as there is a train inside the crossing.
- The controller instructs the gate to open within 1 time unit after receiving an exiting message from the last train leaving the crossing.
- The gate must close within 1 time unit of receiving instructions from the controller.

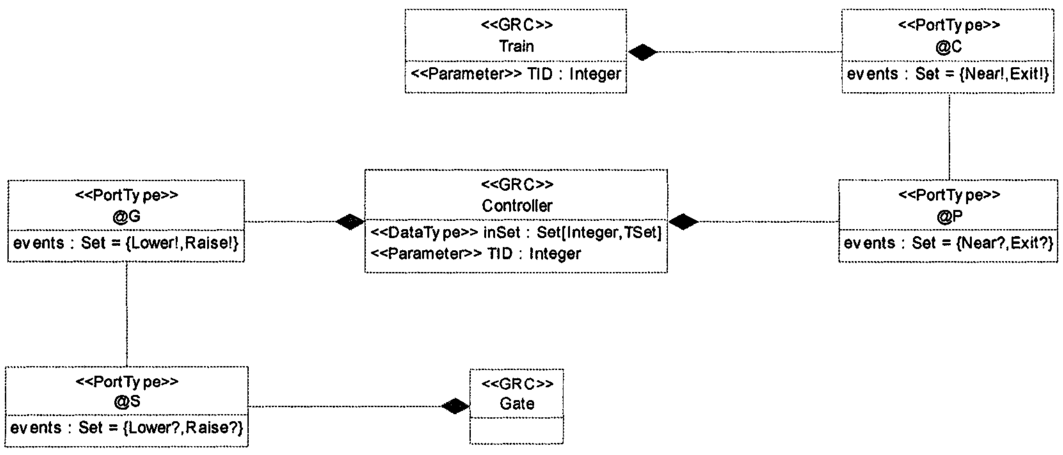
The safety and liveness policies are as following:

Safety:

- Whenever there is a train inside the crossing the gate remains closed

Liveness:

- When the last train leaves the crossing, the gate eventually reopens



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

Figure 11. Class diagram for Railroad case study

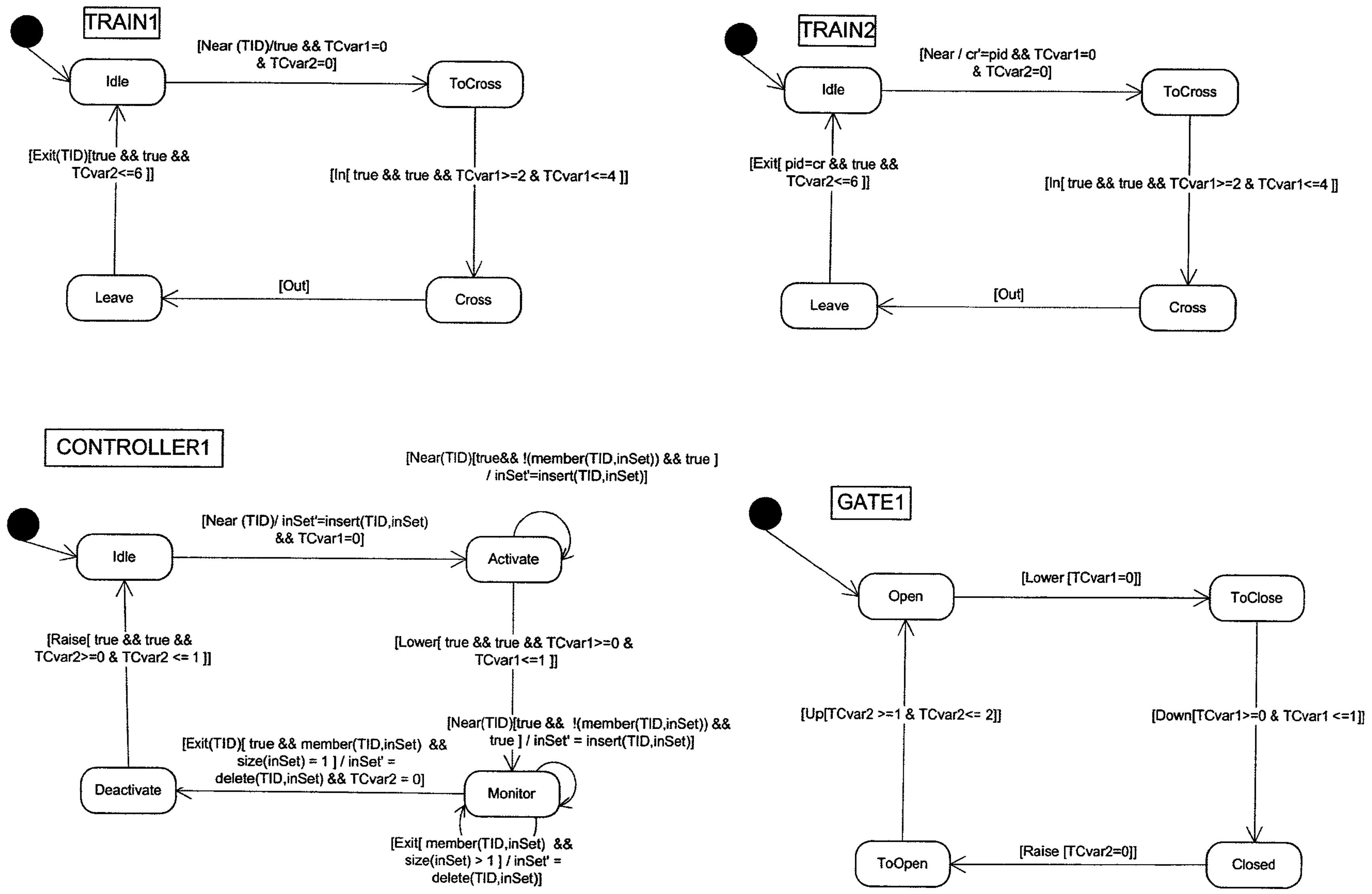

TRAIN1
Idle
ToCross
Cross
Leave
[Near (TID)/true && TCvar1=0 & TCvar2=0]
[In[true && true && TCvar1>=2 & TCvar1<=4]]
[Out]
[Exit(TID)[true && true && TCvar2<=6]]
TRAIN2
Idle
ToCross
Cross
Leave
[Near / cr'=pid && TCvar1=0 & TCvar2=0]
[In[true && true && TCvar1>=2 & TCvar1<=4]]
[Out]
[Exit[pid=cr && true && TCvar2<=6]]
CONTROLLER1
Idle
Activate
Monitor
Deactivate
[Near (TID)/ inSet'=insert(TID,inSet) && TCvar1=0]
[Near(TID)[true&& !(member(TID,inSet)) && true] / inSet'=insert(TID,inSet)]
[Lower[true && true && TCvar1>=0 & TCvar1<=1]]
[Near(TID)[true && !(member(TID,inSet)) && true] / inSet' = insert(TID,inSet)]
[Exit[member(TID,inSet) && size(inSet) > 1] / inSet' = delete(TID,inSet)]
[Exit(TID)[true && member(TID,inSet) && size(inSet) = 1] / inSet' = delete(TID,inSet) && TCvar2 = 0]
[Raise[true && true && TCvar2>=0 & TCvar2 <= 1]]
GATE1
Open
ToClose
Closed
ToOpen
[Lower [TCvar1=0]]
[Down[TCvar1>=0 & TCvar1 <=1]]
[Raise [TCvar2=0]]
[Up[TCvar2 >=1 & TCvar2<= 2]]

Figure 12. State chart diagrams for each reactive object in Railroad case study

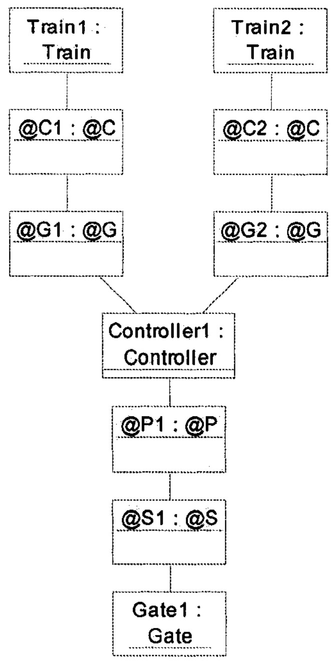


Figure 13. Collaboration diagram for Railroad case study

7.2 Policy Analysis for Railroad Crossing Case Study

Following is the specification of the policies for the case study. Here it is shown how to analyze first the statements in natural language from the case study description and then refine them to form the predicates to be used by the algorithm.

Safety Analysis

- *Whenever there is a train inside the crossing the gate remains closed*

Analyzing the state chart diagrams in fig. 12 it can be deduced that when a train is *inside the crossing* the controller has to be in the state *monitor* and the gate has to be in state *closed.* Actually, the object gate can not issue the event *raise* to open the gate unless the object controller is in state *deactivate.* According to the specification, controller can only transfer from *monitor* to *deactivate* when the last train inside the crossing issues the event *exit.* So, we choose to decide not to take care in avoiding the event *raise* since this can not happen while controller is in state *monitor* and this means train(s) inside the crossing. Our approach is to ensure that no train will be inside the crossing while controller transfers its state from *monitor* to *deactivate.* An object of type train can be in several states inside the crossing but only when it is in state *idle* is considered to be out of the crossing. Consequently, controller can only issue the event *exit* to pass to state *deactivate* only if all but one train are in state *idle* or *tocross*. Since the object controller has two transitions by the same event *exit*, one to transfer to *deactivate* and the other to remain in the same state if the exiting train is not the last one, we have to specify the transition specification in the predicate. The following two safety policies model this behavior:

1. $object\_of(train, idle) \wedge object\_at(controller1, monitor) \wedge object\_at(gate1, closed) \rightarrow$ $\neg\, occur(controller1, exit, R6)$
2. $\neg object\_of(train, idle) \wedge object\_at(controller1, monitor) \wedge object\_at(gate1, closed) \rightarrow$ $\neg\, occur(controller1, exit, R4)$

Policy 1 states that controller can not remain in state *monitor* by issuing transition R6 if all but one object of type train are in state *idle*. Policy 2 stands for the opposite, by restricting a possible issue of transition R4 when not only one object train is in *idle*.

Though not stated in the safety requirements in our case study, we must prevent an object of type train to enter the crossing if the gate is not closed. To do this, we prevent the event *in* from any object of type train in state *tocross* to occur if the gate is in any state where is not closed (*open* and *toclose*). We compose the next policy to address that purpose:

3. $vector\_at(train, tocross) \land object\_at(gate1, open, toclose) \rightarrow \neg\, occur(train, in)$

Liveness Analysis

- *When the last train leaves the crossing, the gate eventually reopens*

When all objects of type train and the object controller are outside the crossing in state *idle* and the object gate is in state closed, gate must issue the event *up*.

4. $vector\_at(train, idle) \land object\_at(controller1, idle) \land object\_at(gate1, toopen) \rightarrow occur(gate1, up)$

A detail of the above liveness policy is that in the same SPM state depicted by its premise, another event may also happen as well which is *near*. To prevent this, we write another safety policy:

5. $vector\_at(train, idle) \land object\_at(controller1, idle) \land object\_at(gate1, toopen) \rightarrow \neg\, occur(controller1, near)$

The final set of policies for the Train-Gate-Controller case study is:

Safety

1. $object\_of(train, idle) \land object\_at(controller1, monitor) \land object\_at(gate1, closed) \rightarrow \neg\, occur(controller1, exit, R6)$
2. $\neg object\_of(train, idle) \land object\_at(controller1, monitor) \land object\_at(gate1, closed) \rightarrow \neg\, occur(controller1, exit, R4)$

3. $vector\_at(train, tocross) \land object\_at(gate1, open, toclose) \rightarrow \neg\, occur(train, in)$
4. $vector\_at(train, idle) \land object\_at(controller1, idle) \land object\_at(gate1, toopen) \rightarrow$ $\neg\, occur(controller1, near)$

Liveness

5. $vector\_at(train, idle) \land object\_at(controller1, idle) \land object\_at(gate1, toopen) \rightarrow$ $occur(gate1, up)$

7.3 Case Study: Railroad Crossing Generalized

The previous case study had a fixed number of objects of type controller and gate. This case study is included to prove the flexibility of the policy specification to a generalized number of objects of any type in the case study. Fig. 14 shows the collaboration diagram from a specification of 2 trains, 2 controllers and 2 gates.

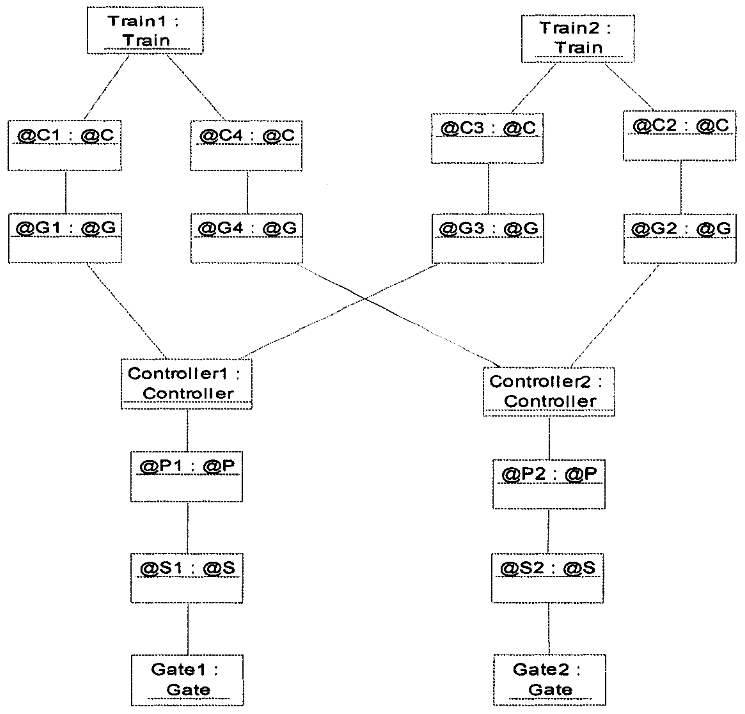


Figure 14. Collaboration diagram for Railroad case study generalized version

7.4 Policy Analysis for Railroad Crossing Case Study Generalized

The same approach is essentially used as in our previous case study. However, there is a change in the specification of an object to its type; there is not a single reference to an specific ARO name. This makes the rules generic for this configuration.

Safety

1. $object\_of(train, idle) \wedge object\_at(controller, monitor) \wedge object\_at(gate, closed) \rightarrow \neg\, occur(controller, exit, R6)$
2. $\neg object\_of(train, idle) \wedge object\_at(controller, monitor) \wedge object\_at(gate, closed) \rightarrow \neg\, occur(controller, exit, R4)$
3. $vector\_at(train, tocross) \wedge object\_at(gate, open, toclose) \rightarrow \neg\, occur(train, in)$

4. $vector\_at(train, idle) \wedge object\_at(controller, idle) \wedge object\_at(gate, toopen) \rightarrow$

   $\neg\, occur(controller, near)$

Liveness

5. $vector\_at(train, idle) \wedge object\_at(controller, idle) \wedge object\_at(gate, toopen) \rightarrow$

   $occur(gate, up)$

7.5 Experimentation and results

A SPM resulting from applying the SPM algorithm to the Train-Gate-Controller case study (2 trains, 1 Controller, 1 Gate) is shown in Appendix A. Because of the amount of information that any other case study issues, only this sample is included of the output of the algorithm. A graphical representation of the SPM can be appreciated from the fig. 15.

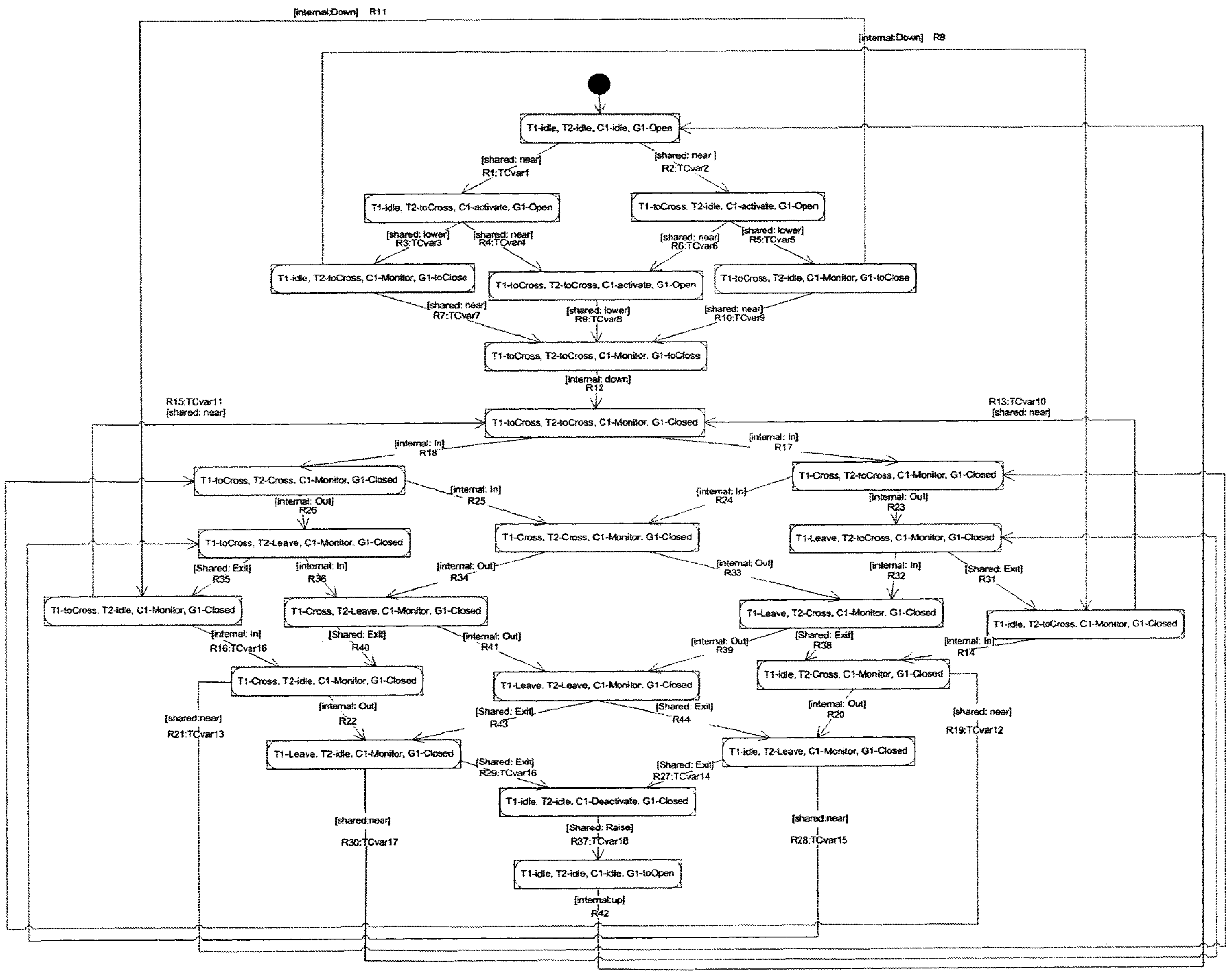

T1-idle, T2-idle, C1-idle, G1-Open
[shared: near] R1:TCvar1
[shared: near] R2:TCvar2
T1-idle, T2-toCross, C1-activate, G1-Open
T1-toCross, T2-idle, C1-activate, G1-Open
[shared: lower] R3:TCvar3
[shared: near] R4:TCvar4
[shared: near] R6:TCvar6
[shared: lower] R5:TCvar5
T1-idle, T2-toCross, C1-Monitor, G1-toClose
T1-toCross, T2-toCross, C1-activate, G1-Open
T1-toCross, T2-idle, C1-Monitor, G1-toClose
[shared: near] R7:TCvar7
[shared: lower] R9:TCvar8
[shared: near] R10:TCvar9
T1-toCross, T2-toCross, C1-Monitor, G1-toClose
[internal: down] R12
[internal:Down] R11
[internal:Down] R8
R15:TCvar11 [shared: near]
R13:TCvar10 [shared: near]
T1-toCross, T2-toCross, C1-Monitor, G1-Closed
[internal: In] R18
[internal: In] R17
T1-toCross, T2-Cross, C1-Monitor, G1-Closed
T1-Cross, T2-toCross, C1-Monitor, G1-Closed
[internal: In] R25
[internal: In] R24
[internal: Out] R26
[internal: Out] R23
T1-toCross, T2-Leave, C1-Monitor, G1-Closed
T1-Cross, T2-Cross, C1-Monitor, G1-Closed
T1-Leave, T2-toCross, C1-Monitor, G1-Closed
[Shared: Exit] R35
[internal: In] R36
[internal: Out] R34
[internal: Out] R33
[internal: In] R32
[Shared: Exit] R31
T1-toCross, T2-idle, C1-Monitor, G1-Closed
T1-Cross, T2-Leave, C1-Monitor, G1-Closed
T1-Leave, T2-Cross, C1-Monitor, G1-Closed
T1-idle, T2-toCross, C1-Monitor, G1-Closed
[internal: In] R16:TCvar16
[Shared: Exit] R40
[internal: Out] R41
[internal: Out] R39
[Shared: Exit] R38
[internal: In] R14
T1-Cross, T2-idle, C1-Monitor, G1-Closed
T1-Leave, T2-Leave, C1-Monitor, G1-Closed
T1-idle, T2-Cross, C1-Monitor, G1-Closed
[shared:near] R21:TCvar13
[internal: Out] R22
[Shared: Exit] R43
[Shared: Exit] R44
[internal: Out] R20
[shared: near] R19:TCvar12
T1-Leave, T2-idle, C1-Monitor, G1-Closed
T1-idle, T2-Leave, C1-Monitor, G1-Closed
[Shared: Exit] R29:TCvar16
[Shared: Exit] R27:TCvar14
T1-idle, T2-idle, C1-Deactivate, G1-Closed
[shared:near] R30:TCvar17
[Shared: Raise] R37:TCvar18
[shared:near] R28:TCvar15
T1-idle, T2-idle, C1-idle, G1-toOpen
[internal:up] R42

Figure 15. State chart diagram showing a produced SPM from railroad case study (version with 2 trains, 2 controller, 1 gate)

The case study depicted in section 6.1 was tested with configurations ranging from 2 to 5 trains. The same set of policies is applied to each new configuration for case study 1. For the second case study the generic rules where used. In all the cases the algorithm terminated without detecting flaws in the policies and the specification. The results of such experimentation are shown in table 1. Additionally, the policies used for the generalized version of the railroad case study in section 6.4 where used for the first case study configuration depicted in fig. 13 yielding the same results in both cases proving that the generalized policies work for any configuration.

	Processing time	Number of iterations	Number of SPM states produced	Number of SPM transitions produced	Number of SPM TC variables produced
2T, 1C, 1G	0.203 sec.	10	24	44	32
3T, 1C, 1G	0.782 sec.	13	80	226	146
4T, 1C, 1G	3.782 sec.	16	288	1112	648
5T, 1C, 1G	26.422 sec.	19	1088	5334	2902
2T, 2C, 2G	9.843 sec.	24	875	3002	2016
3T, 2C, 2G	1 min 42.32 sec.	25	3659	16528	11834

Table 1. Results of testing with different configuration of the Railroad case study

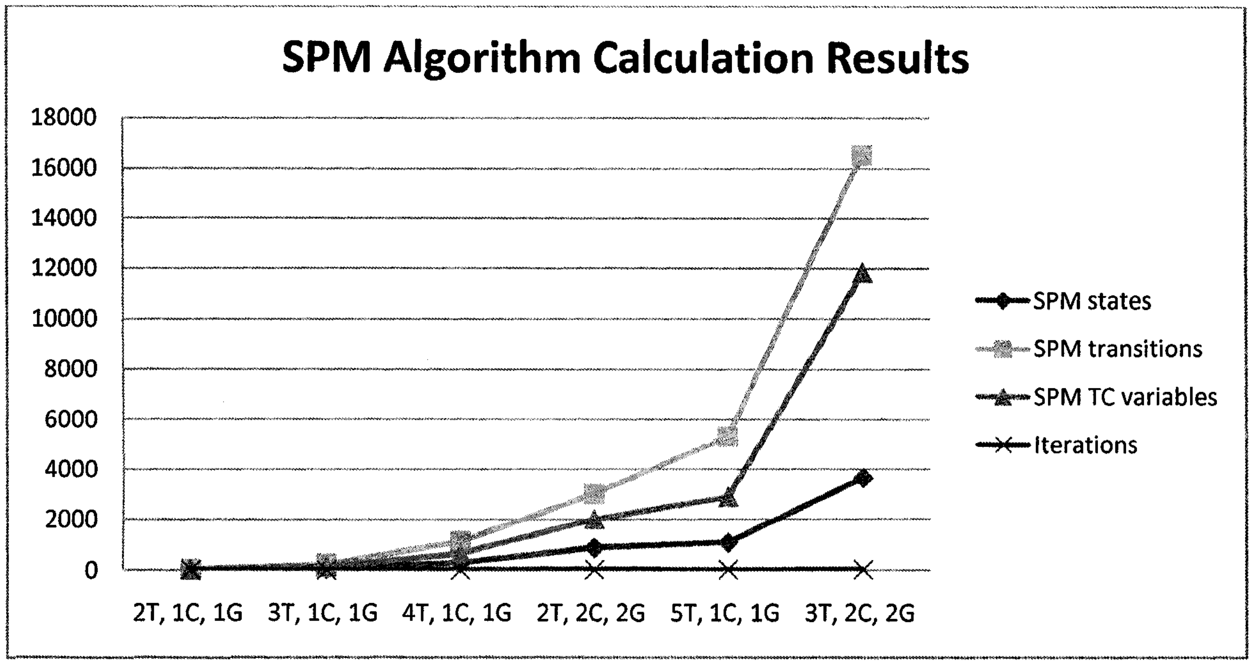


Figure 16. Chart representing the data on table 1

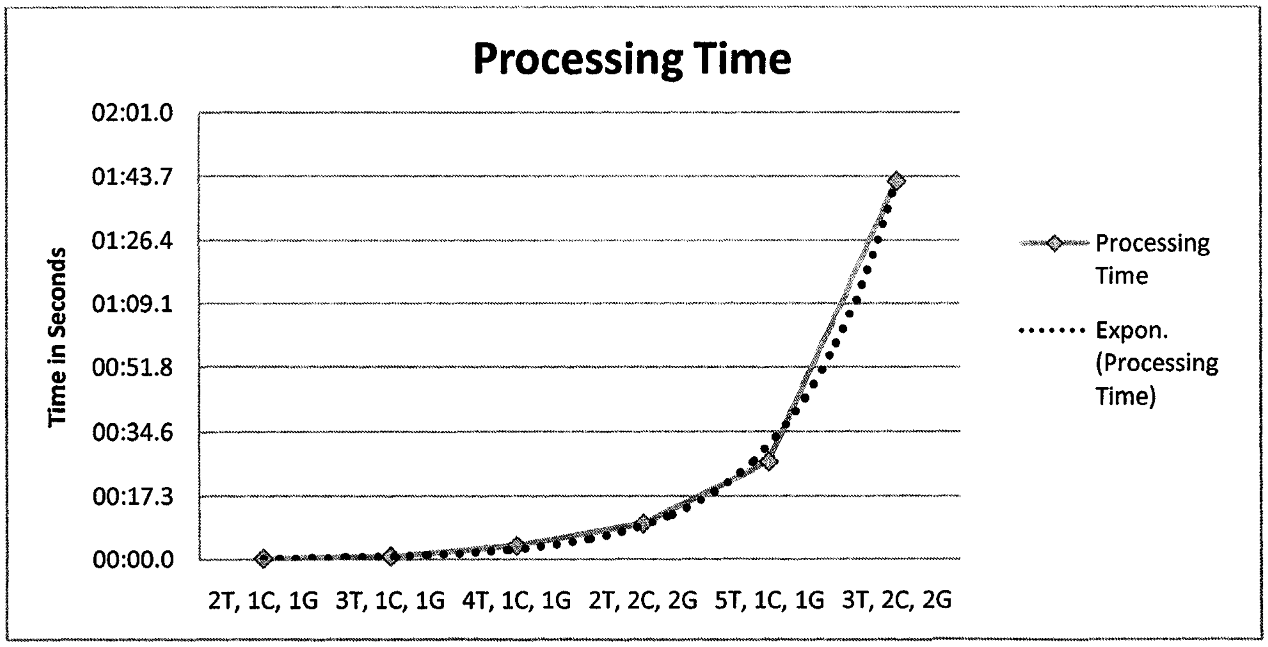


Figure 17. Chart representing execution time of the SPM implementation.

The information exposed in the preceding two charts shows the behavior of the SPM algorithm in various terms. In all the configurations tested the algorithm finished successfully. Looking at the number of SPM states, SPM transitions and SPM time constraint variables generated, we can see how the state explosion increases their numbers as we add more trains to the configuration. The number of SPM transitions is the one that grows more rapidly as more objects are available to create more transitions. The measurement of the processing time is also registered. However, the algorithm is not intended to work at run-time embedded in a real-time system implementation since it would be inappropriate. Rather as early stated, its purpose is to aid at the design stage for validation purposes. The results confirm that the expected growing time behavior of the SPM algorithm as exponential. In fig. 17, an added exponential trend line is used to compare the actual growing time to an exponential one.

Chapter 8. Conclusions and Future Work Directions

8.1 Conclusions

The work of this thesis was mainly concerned with assessing the behavioral correctness of the AS-TRM architecture at component level and to provide an appropriate input for the reliability calculation methodology in AS-TRM. To achieve that, the following tasks where fulfilled:

- Modification and extension of the AS-TRM formalism which included:
  - Modification to the AS-TRM time-constraints formalism for Synchronous Product Machine (SPM) specification.
  - Modification to the AS-TRM time-constraints syntax for Autonomic Reactive Objects (ARO) specification.
- A policy specification was introduced to specify the liveness and safety properties for Autonomic Reactive Components (ARC).
- Design of an algorithm to calculate the SPM of an ARC.
- Design of a methodology to calculate the time constraints of a SPM.
- Implementation and testing of the SPM algorithm with several configurations of the Railroad case study.
- An automatic method has been proposed for verification of the behavioral correctness of an autonomic reactive component against its specifications.

- An automatic method to provide with a data repository represented by the synchronous product machine that can be used to asses the self-control of the behavioral correctness at run-time.
- Design of an algorithm to calculate the minimum and maximum delay time of an ARC and provide insight on the overall performance time of an ARC.

8.2 Future Work

8.2.1 SPM Algorithm

The algorithm proposed in this work is a new contribution to the research of AS-TRM, which proves the concept of self-control of correctness in autonomic reactive systems. However, works in the area of optimization for the algorithm could improve its performance in time. Research in the area of model checking shows that advances in the use of new data structures, like binary decision diagrams, makes it possible to manipulate data structures faster and more efficiently [13]. Also, several methods and techniques have been proposed to deal with the state explosion problem like partial order reduction, symbolic representation, symmetry and others [13]. These techniques could be applied to produce a better and more efficient algorithm design.

8.2.2 Self-Monitoring in AS-TRM

Because the SPM algorithm provides with a data repository representing the behavior of an autonomic reactive component it can aid in its self-monitoring feature. Appropriate mechanisms needs to be created to take advantage of the information created by the algorithm.

8.2.3 Minimum and Maximum Delay Algorithm implementation

The minimum and maximum delay algorithm proposed in this thesis is not implemented yet. This algorithm might be suitable in a research work context regarding planning and estimation of resources in AS-TRM.

Bibliography

[1] P. Horn, "Autonomic Computing: IBM Perspective on the State of Information Technology," IBM T. J. Watson Laboratories, October, 2001.

[2] A. Klein, "Optimizing Enterprise IT. Intelligent Enterprise", February, 2005, http://www.intelligententerprise.com/showarticle.jhtml?articleID=60403261.

[3] M. Parashar and S. Hariri, "Autonomic Computing: Concepts, Infrastructure and Applications," Boca Raton, CRC press, 2007.

[4] V. Alagar, R. Achuthan and D. Muthiayen, "TROMLAB: A Software Development Environment for Real-Time Reactive Systems", Technical Report , (first version 1996, revised 2001), Department of Computer Science , Concordia University.

[5] J.V. Guttag, J.J. Horning, "Larch: Languages and Tools for Formal Specification," Springer-Verlag, January, 1993.

[6] H. Kuang, "Architecture for Reactive Autonomic Systems. AS-TRM Approach," M.Sc thesis, Computer Science and Software Engineering Department, Concordia University, Montreal, QC, Canada, 2006.

[7] Emil Vassev, Olga Ormandjieva, Joey Paquet. "ASSL Specification of Reliability Self-Assessment in the AS-TRM", In Proceedings of the 2nd International Conference on Software and Data Technologies (ICSOFT 2007), pp. 198-206, Barcelona, Spain, July 2007.

[8] R, Achuthan, "A Formal Model for Object-Oriented Development of Real-Time Reactive Systems", M.Sc thesis, Concordia University, Montreal, QC, Canada, 2006.

[9] O. Ormandjieva, H. Kuang, E. Vassev. "Reliability Self-Assessment in Reactive Autonomic Systems: Autonomic System-Time Reactive Model Approach".

International Transactions on Systems Science and Applications, Volume 2 (1), pp.99-104, 2006.

[10] ISO/IEC 19761. Software Engineering – COSMIC-FFP – A functional size measurement method, International Organization for Standardization – ISO, Geneva, 2003.

[11] D. Muthiayen, "Animation and Formal Verification of Real-Time Reactive Systems in an Object-oriented Environment", M.Sc thesis, Concordia University, Montreal, QC, Canada, 1996.

[12] V. Alagar, O. Ormandjieva and M. Zheng "Managing Complexity in Real-Time Reactive Systems". Concordia University, Montreal, QC, Canada, 2001.

[13] E. Clarke, " Model Checking", Prentice-Hall, 1999

[14] C. Shaffer, "A Practical Introduction to Data Structures and Algorithm Analysis", Prentice Hall, 1998.

[15] V. Alagar, K. Periyasami "Specification of Software Systems", Springer, New York, 1998.

[16] Manar Abu-Talib, Olga Ormandjieva, Alain Abran. "AS-TRM and Functional Size with COSMIC-FFP", In the Proceedings of the IEEE International Symposium on Industrial Electronics – ISIE 2007, Vigo, Spain, June 4-7, 2007.

[17] Irina Paltin. "Autonomic Systems Modeling and Development: A Survey", M.S. Major Report , Computer Science and Software Engineering Department, Concordia University, Montreal, QC, Canada, 2006.

[18] Olga Ormandjieva, Emil Vassev. "Towards ASSL Specification of Self-Scheduling Design and Monitoring in Team-Robotics Modeled with AS-TRM", In Proceedings

of the IEEE Engineering/Computing and Systems Research E-Conference (CIISE 2007), December 3 - 12, 2007, University of Bridgeport.

[19] O. Ormandjieva. "Modeling and Monitoring NFRs in Autonomic Systems: AS-TRM Approach". In Proceedings of the XL International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST 2005), pp. 683-686, June 19-22, 2005.

Appendix A

This section presents the results of the SPM algorithm with the configuration: 2 trains, 1 controller and 1 gate.

SPM states:

[idle, open, idle, idle], [activate, open, idle, tocross], [activate, open, tocross, idle],

[monitor, toclose, idle, tocross], [activate, open, tocross, tocross], [monitor, toclose, tocross, idle],

[monitor, toclose, tocross, tocross], [monitor, closed, idle, tocross], [monitor, closed, tocross, idle],

[monitor, closed, tocross, tocross], [monitor, closed, idle, cross], [monitor, closed, cross, idle],

[monitor, closed, cross, tocross], [monitor, closed, tocross, cross], [monitor, closed, idle, leave],

[monitor, closed, leave, idle], [monitor, closed, leave, tocross], [monitor, closed, cross, cross],

[monitor, closed, tocross, leave], [deactivate, closed, idle, idle], [monitor, closed, leave, cross],

[monitor, closed, cross, leave], [idle, toopen, idle, idle], [monitor, closed, leave, leave]

SPM transitions:

R1 , [idle, open, idle, idle] , [activate, open, idle, tocross] , near , (true) , true=>inSet'=insert(pid,inSet);

R2 , [idle, open, idle, idle] , [activate, open, tocross, idle] , near , (true) , true=>inSet'=insert(pid,inSet);

R3 , [activate, open, idle, tocross] , [monitor, toclose, idle, tocross] , lower , (true) , true=>true;

R4 , [activate, open, idle, tocross] , [activate, open, tocross, tocross] , near , (!(member(pid,inset))) ,

true=>inSet'=insert(pid,inSet);

R5 , [activate, open, tocross, idle] , [monitor, toclose, tocross, idle] , lower , (true) , true=>true;

R6 , [activate, open, tocross, idle] , [activate, open, tocross, tocross] , near , (!(member(pid,inset))) ,

true=>inSet'=insert(pid,inSet);

R7 , [monitor, toclose, idle, tocross] , [monitor, toclose, tocross, tocross] , near , (!(member(pid,inset))) ,

true=>inSet'=insert(pid,inSet);

R8 , [monitor, toclose, idle, tocross] , [monitor, closed, idle, tocross] , down , (true) , true=>true;

R9 , [activate, open, tocross, tocross] , [monitor, toclose, tocross, tocross] , lower , (true) , true=>true;

R10 , [monitor, toclose, tocross, idle] , [monitor, toclose, tocross, tocross] , near , (!(member(pid,inset))) , true=>inSet'=insert(pid,inSet);

R11 , [monitor, toclose, tocross, idle] , [monitor, closed, tocross, idle] , down , (true) , true=>true;

R12 , [monitor, toclose, tocross, tocross] , [monitor, closed, tocross, tocross] , down , (true) , true=>true;

R13 , [monitor, closed, idle, tocross] , [monitor, closed, tocross, tocross] , near , (!(member(pid,inset))) , true=>inSet'=insert(pid,inSet);

R14 , [monitor, closed, idle, tocross] , [monitor, closed, idle, cross] , in , true=>true ,

R15 , [monitor, closed, tocross, idle] , [monitor, closed, tocross, tocross] , near , (!(member(pid,inset))) , true=>inSet'=insert(pid,inSet);

R16 , [monitor, closed, tocross, idle] , [monitor, closed, cross, idle] , in , true=>true ,

R17 , [monitor, closed, tocross, tocross] , [monitor, closed, cross, tocross] , in , true=>true ,

R18 , [monitor, closed, tocross, tocross] , [monitor, closed, tocross, cross] , in , true=>true ,

R19 , [monitor, closed, idle, cross] , [monitor, closed, tocross, cross] , near , (!(member(pid,inset))) , true=>inSet'=insert(pid,inSet);

R20 , [monitor, closed, idle, cross] , [monitor, closed, idle, leave] , out , true=>true ,

R21 , [monitor, closed, cross, idle] , [monitor, closed, cross, tocross] , near , (!(member(pid,inset))) , true=>inSet'=insert(pid,inSet);

R22 , [monitor, closed, cross, idle] , [monitor, closed, leave, idle] , out , true=>true ,

R23 , [monitor, closed, cross, tocross] , [monitor, closed, leave, tocross] , out , true=>true ,

R24 , [monitor, closed, cross, tocross] , [monitor, closed, cross, cross] , in , true=>true ,

R25 , [monitor, closed, tocross, cross] , [monitor, closed, cross, cross] , in , true=>true ,

R26 , [monitor, closed, tocross, cross] , [monitor, closed, tocross, leave] , out , true=>true ,

R27 , [monitor, closed, idle, leave] , [deactivate, closed, idle, idle] , exit , (member(pid,inset)) , size(inSet)=1=>inSet'=delete(pid,inSet);

R28 , [monitor, closed, idle, leave] , [monitor, closed, tocross, leave] , near , (!(member(pid,inset))) , true=>inSet'=insert(pid,inSet);

R29 , [monitor, closed, leave, idle] , [deactivate, closed, idle, idle] , exit , (member(pid,inset)) ,

size(inSet)=1=>inSet'=delete(pid,inSet);

R30 , [monitor, closed, leave, idle] , [monitor, closed, leave, tocross] , near , (!(member(pid,inset))) ,

true=>inSet'=insert(pid,inSet);

R31 , [monitor, closed, leave, tocross] , [monitor, closed, idle, tocross] , exit , (member(pid,inset)) ,

size(inSet)>1=>inSet'=delete(pid,inSet);

R32 , [monitor, closed, leave, tocross] , [monitor, closed, leave, cross] , in , true=>true ,

R33 , [monitor, closed, cross, cross] , [monitor, closed, leave, cross] , out , true=>true ,

R34 , [monitor, closed, cross, cross] , [monitor, closed, cross, leave] , out , true=>true ,

R35 , [monitor, closed, tocross, leave] , [monitor, closed, tocross, idle] , exit , (member(pid,inset)) ,

size(inSet)>1=>inSet'=delete(pid,inSet);

R36 , [monitor, closed, tocross, leave] , [monitor, closed, cross, leave] , in , true=>true ,

R37 , [deactivate, closed, idle, idle] , [idle, toopen, idle, idle] , raise , (true) , true=>true;

R38 , [monitor, closed, leave, cross] , [monitor, closed, idle, cross] , exit , (member(pid,inset)) ,

size(inSet)>1=>inSet'=delete(pid,inSet);

R39 , [monitor, closed, leave, cross] , [monitor, closed, leave, leave] , out , true=>true ,

R40 , [monitor, closed, cross, leave] , [monitor, closed, cross, idle] , exit , (member(pid,inset)) ,

size(inSet)>1=>inSet'=delete(pid,inSet);

R41 , [monitor, closed, cross, leave] , [monitor, closed, leave, leave] , out , true=>true ,

R42 , [idle, toopen, idle, idle] , [idle, open, idle, idle] , up , (true) , true=>true;

R43 , [monitor, closed, leave, leave] , [monitor, closed, leave, idle] , exit , (member(pid,inset)) ,

size(inSet)>1=>inSet'=delete(pid,inSet);

R44 , [monitor, closed, leave, leave] , [monitor, closed, idle, leave] , exit , (member(pid,inset)) ,

size(inSet)>1=>inSet'=delete(pid,inSet);

SPM Time Constraints:

TCvar1: R1, SPM: [lower.R3, lower.R5, lower.R9], lower.r1.controller1;[0,1];{ };

TCvar1: R1, SPM: [exit.R27, exit.R35, exit.R40, exit.R43], exit.r3.train2;[0,6];{ };

TCvar1: R1, SPM: [in.R14, in.R18, in.R24, in.R32], in.r4.train2;[2,4];{ };

TCvar2: R2, SPM: [lower.R3, lower.R5, lower.R9], lower.r1.controller1;[0,1];{ };

TCvar2: R2, SPM: [exit.R29, exit.R31, exit.R38, exit.R44], exit.r3.train1;[0,6];{ };

TCvar2: R2, SPM: [in.R16, in.R17, in.R25, in.R36], in.r4.train1;[2,4];{ };

TCvar3: R3, SPM: [down.R8, down.R11, down.R12], down.r2.gate1;[0,1];{closed};

TCvar4: R4, SPM: [exit.R29, exit.R31, exit.R38, exit.R44], exit.r3.train1;[0,6];{ };

TCvar4: R4, SPM: [in.R16, in.R17, in.R25, in.R36], in.r4.train1;[2,4];{ };

TCvar5: R5, SPM: [down.R8, down.R11, down.R12], down.r2.gate1;[0,1];{closed};

TCvar6: R6, SPM: [exit.R27, exit.R35, exit.R40, exit.R43], exit.r3.train2;[0,6];{ };

TCvar6: R6, SPM: [in.R14, in.R18, in.R24, in.R32], in.r4.train2;[2,4];{ };

TCvar7: R7, SPM: [exit.R29, exit.R31, exit.R38, exit.R44], exit.r3.train1;[0,6];{ };

TCvar7: R7, SPM: [in.R16, in.R17, in.R25, in.R36], in.r4.train1;[2,4];{ };

TCvar8: R9, SPM: [down.R11, down.R12], down.r2.gate1;[0,1];{closed};

TCvar9: R10, SPM: [exit.R27, exit.R35, exit.R40, exit.R43], exit.r3.train2;[0,6];{ };

TCvar9: R10, SPM: [in.R14, in.R18, in.R24, in.R32], in.r4.train2;[2,4];{ };

TCvar10: R13, SPM: [exit.R29, exit.R31, exit.R38, exit.R44], exit.r3.train1;[0,6];{ };

TCvar10: R13, SPM: [in.R16, in.R17, in.R25, in.R36], in.r4.train1;[2,4];{ };

TCvar11: R15, SPM: [exit.R27, exit.R35, exit.R40, exit.R43], exit.r3.train2;[0,6];{ };

TCvar11: R15, SPM: [in.R18, in.R24, in.R32], in.r4.train2;[2,4];{ };

TCvar12: R19, SPM: [exit.R29, exit.R31, exit.R38, exit.R44], exit.r3.train1;[0,6];{ };

TCvar12: R19, SPM: [in.R25, in.R36], in.r4.train1;[2,4];{ };

TCvar13: R21, SPM: [exit.R27, exit.R35, exit.R40, exit.R43], exit.r3.train2;[0,6];{ };

TCvar13: R21, SPM: [in.R24, in.R32], in.r4.train2;[2,4];{ };

TCvar14: R27, SPM: [raise.R37], raise.r3.controller1;[0,1];{ };

TCvar15: R28, SPM: [exit.R29, exit.R31, exit.R38, exit.R44], exit.r3.train1;[0,6];{ };

TCvar15: R28, SPM: [in.R36], in.r4.train1;[2,4];{ };

TCvar16: R29, SPM: [raise.R37], raise.r3.controller1;[0,1];{ };

TCvar17: R30, SPM: [exit.R35, exit.R40, exit.R43], exit.r3.train2;[0,6];{ };

TCvar17: R30, SPM: [in.R32], in.r4.train2;[2,4];{ };

TCvar18: R37, SPM: [up.R42], up.r3.gate1;[1,2];{ };