TRANSFORMING ARCHITECTURAL DESCRIPTIONS OF COMPONENT-BASED SYSTEMS FOR FORMAL ANALYSIS

NASEEM ISMAIL IBRAHIM

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

DECEMBER 2008

Your file Votre référence
ISBN: 978-0-494-63331-1
Our file Notre référence
ISBN: 978-0-494-63331-1

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Naseem Ismail Ibrahim**

Entitled: **Transforming Architectural Descriptions of Component-based Systems for Formal Analysis**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair

Dr. Peter Grogono

_____ Examiner

Dr. Sabine Bergler

_____ Examiner

Dr. Olga Ormandjieva

_____ Supervisor

Dr. Vangalur Alagar

Approved _____

Chair of Department or Graduate Program Director

_____ 20 _____ _____

Dr. Robin A.L. Drew, Dean

Faculty of Engineering and Computer Science

ABSTRACT

Transforming Architectural Descriptions of Component-based Systems for Formal Analysis

Naseem Ismail Ibrahim

Design time analysis is an important step in the process of developing software systems, with the goal of ensuring that the system design conforms to the design constraints that are stated as part of the functional and non-functional requirements. The well-known techniques for formally analyzing a design are model checking, axiom-based formal verification, and real-time schedulability analysis that takes into account resource constraints. In this thesis, model checking and real-time schedulability are the techniques used to verify that the system under development is both safe and secure.

The architecture of a trustworthy system, formally described in Trustworthy Architectural Description Language (TADL), is taken as the input for the analysis stage. Instead of developing new tools to perform the analyses, the thesis has developed transformation techniques to transform TADL descriptions into behaviour protocols used by existing verification tools. The transformation rules are described independently of the transformation process, thus allowing both reuse and easy extendability. A tool based on such techniques has been designed and implemented which automatically generates two types of models from a TADL description. One is the UPPAAL model, on which the security and safety properties of the system under design are formally verified. The second output is the TIMES model, on which real-time schedulability analysis is performed. The techniques and tools are applied to *The Common Component Modelling Example* (CoCoME), a case study defined by the component development community, to demonstrate that TADL is expressive enough to formally describe component-based systems.

ACKNOWLEDGMENTS

First, I would like to show my profound thanks and appreciation to my supervisor Dr. Alagar for his guidance and support. This work would not exist without him.

Furthermore, I would like to acknowledge and appreciate the help and support of my friend and colleague Mubarak.

Finally, and most importantly, I would like to thank my parents, who provided the item of greatest worth - opportunity. Thank you for standing by me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Research in the development of *Trustworthy Computing Systems* (TCS) is relatively new. In January of 2002, Microsoft published a paper on *Trustworthy Computing*. In October of 2002, a revised version of this paper was made available on the Web [MdVHC02]. One can safely say that this paper created immense interest in the research community. Our research group began work on TCS in 2005. My thesis is a small contribution to the extensive and on-going studies being carried out by our research group.

As humans, we tend to trust a technology when it becomes so dependable that we are oblivious to its internal technical details as it operates in our daily lives. The automobile, electricity and the telephone are classic examples of trusted technologies which have been broadly adopted in our world, because they work as advertised and they are there without fail when we need them. By comparison, although computers and computing services have become commonplace, we are not yet in a position to trust them. The computing paradigm, in spite of its pervasive nature, has not changed in the last 30 or 40 years. The TCS initiative is about to change that paradigm through a combination of engineering principles, business practices and regulatory service provision. Abstracted, these are the principles governing *safety*, *reliability*, and *business integrity*. The challenge is to ensure that the qualities associated with these principles are inherent in the artifacts produced throughout the various stages of the software development process, and the procedures followed during the deployment of the system and its management when it is operational. The TCS

research group, working under the supervision of Dr. Alagar, is undertaking research in the *component-based development* of TCS, which involves the following:

1. A rigorous process model, with a formal architecture and its description
2. A formal definition of trustworthiness properties
3. A formal definition of trustworthy components, and their composition
4. Languages and methods for specification and design
5. Formal techniques for design-time validation and verification
6. Rigorous methods for implementation, reuse, and deployment
7. A framework for developing TCS based on the above techniques

This thesis constitutes a contribution to framework development. It focuses on the design-time analysis to ensure that the system design confirms to the design constraints that are stated as part of the functional and non-functional requirements.

1.1 Trustworthy Computing System Development

Alagar and Mubarak [AM07a] have proposed a component-based development approach for TCS which is also a real-time reactive system (RTRS) approach. An un-timed TCS is a special case of a trustworthy RTRS. In this section, we give a quick summary of the RTRS, we define the TCS and component models, and we describe the development approach proposed in [MA08a].

1.1.1 Real-Time Reactive System (RTRS)

Reactive systems interact with their environment in a continuous and ongoing way. They are event-driven, they interact intensively with their environment through stimulus response behavior, and they are regulated by strict timing constraints. Furthermore, these systems

might also consist of both physical and software components which continuously control the physical devices. Although reactive systems are also interactive, they are fundamentally different from interactive systems. Whereas both environment and processes have synchronization abilities in interactive systems, a process in a reactive system is solely responsible for the synchronization of its environment. That is, a process in a reactive system is fast enough to react to a stimulus from the environment, and the time between stimulus and response is sufficient for the dynamics of the environment to be receptive to the response. For example, a human-computer interface is an interactive system, whereas a controller that regulates the amount of steam escaping from a boiler is clearly reactive. In the case of real-time reactive systems, stimulus-response behaviour is also regulated by timing constraints, and the major design issue is performance. Examples of RTRS include telephony, air traffic control systems, nuclear power reactors and avionics. The major factors that contribute to the complexity of RTRS are the following:

- *size:* telephony and air traffic control systems are made up of a large number of hardware and software components;
- *time constraints:* telephony imposes only *soft* time constraints, a violation of which may not lead to a catastrophe, but may affect user trust; however, avionics and nuclear power control systems impose *hard* (strict) time constraints, which, if violated, will cause damage and injury to human safety, and perhaps shatter user trust entirely;
- *criticality:* a nuclear power reactor is a safety-critical system, in the sense that its failure would be unacceptable;
- *heterogeneity:* sensors, actuators and system processes have different levels of functional and time-sensitive synchronization requirements.

It is evident from the above discussion that RTRS must be trustworthy, and that the essential features of the RTRS that determine its trustworthiness are safety and reliability.

1.1.2 Trustworthiness Credentials

In general, trust is a social concept that is hard to define formally. However, in the software industry [ALR01, SBI99], there is a consensus on its definition. In the software development community, the terms *trustworthiness* and *dependability* are used interchangeably. Trustworthiness is the system property that denotes the degree of user confidence that the system will behave as expected [SBI99, CL02]. Dependability is defined as the ability to deliver trusted services [ALR01]. A comparison of the two terms presented in [ALR01] has concluded that the they are equivalent in their goals and address similar concerns. The goals of dependability are: (1) to provide justifiably trusted services; and (2) to avoid service outage that is unacceptable to the consumer. Thus, dependability and trustworthiness involve achieving *availability*,*reliability*, *safety*, *security* and *survivability*. Safety and security are non-functional requirements which can be formally specified as system properties at design time. Reliability is a quality to be measured while the system is operational. System availability and survivability are to be assessed in an operational environment under different load factor assumptions, and patterns of attacks on the security of the system. If safety and security are assured, they will eventually ensure a higher rate of system availability and reliability. Therefore, in [AM07a], they have considered safety and security as the essential credentials of trustworthiness during the design stage, and reliability and survivability as properties to be assessed after implementation of the system. For the latter, the framework provides a comprehensive set of tools in the implementation and run-time environment of the system.

1.1.3 Component-based Development (CBD)

CBD is the type of software engineering development in which systems are built by constructing units, called components, that perform simple tasks, and assembling them to create composite components that perform complex tasks. Some potential benefits of applying CBD for RTRS include complexity reduction, time and cost savings, predictable behaviour and productivity increase [CL02]. In [AM07a], a review of the literature on CBD is given.

Their discussion points out the inconsistency in existing component definitions and a need to define components more formally. They have proposed a component model which collectively addresses the requirements of RTRS and the credentials of trustworthiness. A central challenge in building trustworthy systems using the CBD method is composing trustworthy components such that each individual component is trustworthy. Their main contributions in [AM07a] are: (1) a definition of the requirements of a component model for developing trustworthy RTRS; (2) a formal definition for trustworthy hierarchical RTRS components; and (3) a compositional theory for composing components so that safety and security are preserved in the composition. Because the component definition is richer and more intricate than existing component definitions, it is not possible to adapt the existing tools in an implementation of new trustworthy component models. This sets the stage for discussing the scope and contribution of this thesis.

1.1.4 Scope of the Thesis

The primary goal of the development framework is to provide a basis for the rigorous development, analysis and deployment of TCS. The application developer, who is normally an expert in the application domain, should be permitted to focus on the modeling and analysis aspects without being burdened by the formalism. That formalism should be working in the background, ensuring that nothing improper is done in the construction of the system. Another goal is to reduce the complexity in the process of understanding the results and behaviour of the system through the introduction of task-oriented descriptions in the development framework which are easy to use and easy to learn. These are the strong motivations behind the development of a Visual Modeling Tool. From a visual model, the formal behaviour model is automatically generated and formally verified for trustworthiness properties. Using the Real-time View, a formal schedulability analysis is performed. In this thesis, behaviour model generation and schedulability analysis are the two primary goals. The architecture of a trustworthy system, formally described in Trustworthy Architectural Description Language (TADL), is taken as the input for the analysis stage. Instead of developing new tools to perform the analyses, the thesis has developed transformation

techniques to transform TADL descriptions into behaviour protocols used by existing verification tools. A tool based on such techniques has been designed and implemented. The tool automatically generates two types of models from a TADL description. One is the UPPAAL model, on which security and safety properties of the system under design are formally verified. The second output is the TIMES model, on which real-time schedulability analysis is performed.

Chapter 2 discuses the TADL, UPPAAL and TIMES models representing the input and output units for the transformation process. A detailed description of the transformation process, rules and algorithm are presented in Chapter 3, which also contains the transformation tool design. The transformation tool implementation is presented in Chapter 4. Two case studies are presented in Chapter 5, and tested using the techniques and tool presented in this thesis. Finally, Chapter 6 provides the conclusion of this thesis and directions for future research.

Chapter 2

Basic Concepts

In Chapter 1, a brief introduction to the component-based model for devolving trustworthy real-time reactive systems (TADL model) has been given and the contribution of the work done in this thesis discussed. This thesis is concerned with the design-time analysis of a system designed using our component model. The design-time analysis includes model checking and real-time analysis. This Chapter presents a more detailed discussion of the basic concepts on which the rest of the thesis depends. Section 1 discusses the meta-model of our component-based model, which is defined in [MA08b] and reviewed here to provide a broader understanding of the model and its main elements. This will help in understanding the process and the rules applied in transforming a system defined using this component model to the UPPAAL and TIMES models. Section 2 discusses the UPPAAL model checker tool, which will be used to check the behaviour of the system defined using our component model, as well as the UPPAAL architecture, which will be of major importance when discussing the transformation of systems from our component model to the UPPAAL model. Section 3 presents the TIMES tool, the TIMES architecture and input language.

2.1 Meta-model for Component-based Trustworthy Real-time Reactive System

In Chapter 1, a brief introduction to the component-based model for developing real-time reactive systems (TADL model) has been introduced. This section will provide a more detailed discussion by reviewing the meta-model of this component model. A meta-model is a model that explains a set of related models [Ama04]. It is used to model architectures by defining types from which system architecture can be defined. Figure 1, which is taken from [MA08b], shows the meta-model for our component model. The main elements of this meta-model are described below.

2.1.1 Component definition

A *component* is an instance of a *component type*. A component type is an aggregation of *interface types*. An interface type is an access point through which *services* can be provided or requested. A service is a functionality provided by or required by a component. Each service can be provided by only one interface, which is an instance of an interface type. A service can also have multiple *data parameters*. A data parameter is a variable passed on to a component within a request for a service or passed on with a provided service.

2.1.2 Architecture definition

A component type can be *primitive* or *composite*. A composite component type is constructed from multiple component types connected to one another using *connector types*. An *architecture* defines the structure of a composite component, it defines the components of which it consists and the internal connection between them. A component type can have one or more architectures associated with it. Components initialized from component types are connected in the architecture. This connection is made by connecting their interface types to different *connector roles* of the same *connector*. A connector defines the

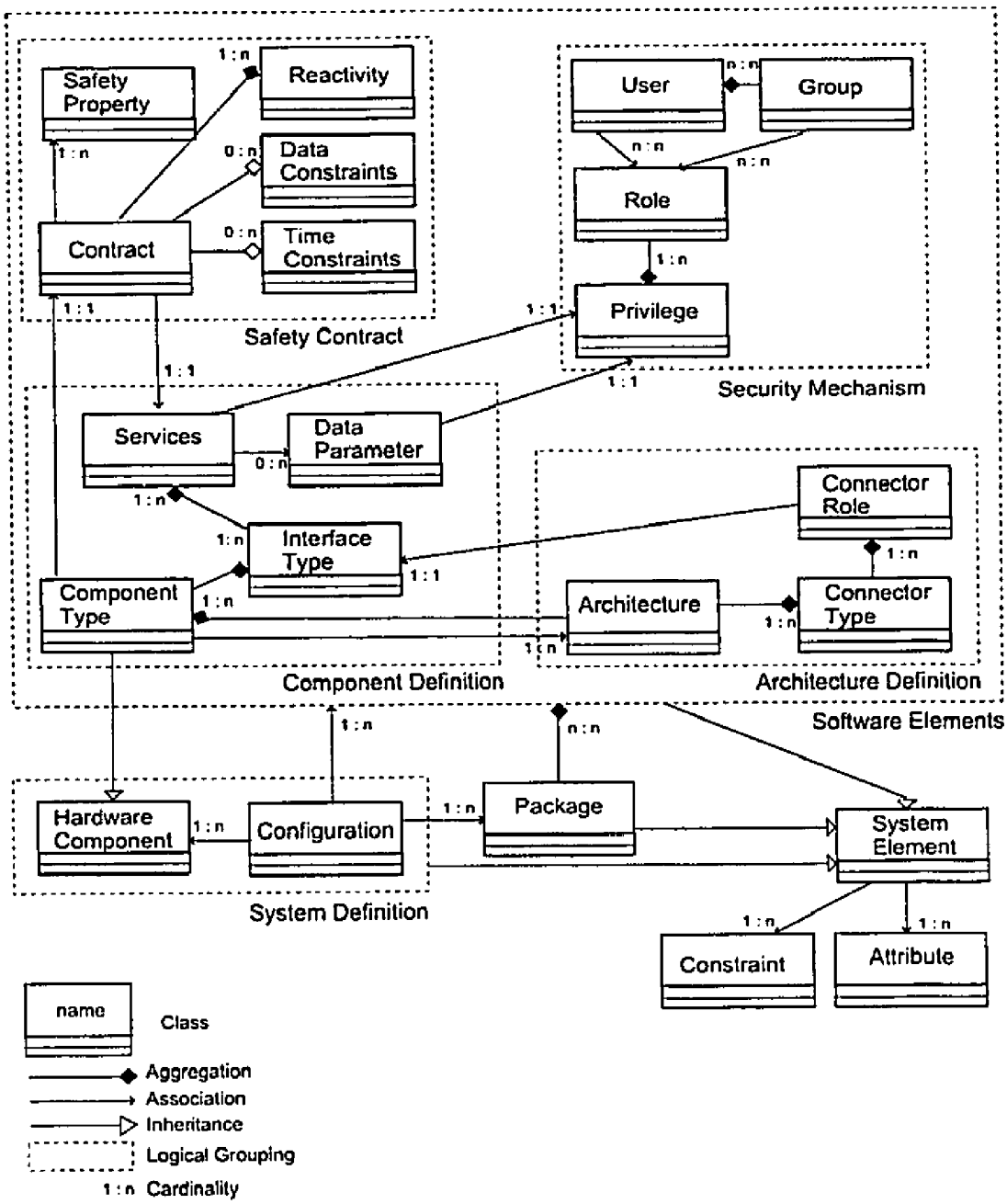

Figure 1: Meta-model

connectivity between components. A connector role defines an access point of communication. The connector role serves as an interface to a connector. A primitive component has no architecture.

2.1.3 Safety contract

A safety property is considered a *contract* on a component type. It controls the way services are provided or requested. Each contract has a one-to-one relationship with a component type. A contract can have one or more *safety properties*, where a safety property defines an invariant over the component behaviour. Each contract should contain at least one *reactivity* property, each of which expresses a relationship between a *stimulus* (request) and a *response*. A contract may have an optional *data constraint* and an optional *time constraint*. A data constraint is a condition that must be satisfied to enable the reactivity property; in other words, it is a constraint that should be checked to decide whether to send a specific response to a request or not. Data constraints are defined over the values of the data parameters of the stimulus service. It is possible for the same stimulus to have multiple responses. It is also possible to define data constraints which are mutually exclusive for each response, and so, depending on the results of evaluating those data constraints, only one response is chosen. A time constraint defines the maximum allowed time between receiving a request and providing a response. An example of a time constraint would be: *a response must be sent within 4 seconds of receiving the request.*

2.1.4 Security mechanism

A security property is concerned with controlling access to the services and the data communicated with those services. The same *security mechanism* can be associated with several component types, and there is a one-to-many relationship between a security mechanism and component types. *Role-based access control* is currently enforced as the only security mechanism, and has the following main elements: *user*, *group*, *role* and *privilege*. A user defines the identity on behalf of which the component will execute. A group is

a collection of users, and a user may belong to more than one group. A role defines the responsibilities that can be assumed by a user or a group in the system. A role aggregates a set of privileges, where each privilege defines a permission to perform a service or access a data parameter. Security can be divided into two types: *service security* and *data security*. Service security is concerned with securing the services provided by the components. It does that by ensuring that:

- every request received at a component interface is initialized by a user who has permission to request this service - if the user has no access permission, the request is ignored;
- the user of a response sent from a component interface has permission to receive that response - if the user has no access permission, the response will not be sent.

Data security is concerned with securing the data transferred with services. This is done by ensuring that:

- the person initializing a request at a component interface has permission to access all the data parameters associated with that request - if the user has to access permission to the data parameters, their values are set to null;
- the user receiving a response from a component interface has permission to access all its data parameters - if the user has to access permission to the data parameters, their values are set to null.

2.1.5 System definition

System definition consists of: *hardware components* and *configuration*. Definition of the hardware components includes specification of the deployment units that will host software components. The configuration definition contains the system specifications, including:

- description of the hardware and software components with the initialization of their local attributes;

- specification of the system users and their assignment to security groups and roles;
- the deployment specification, which states the hardware component to which each software component is deployed.

2.1.6 Attribute, Constraint and Package

An *attribute* is used to define semantic information which can be associated with any meta-model element using typed, named values. A *constraint* is used to define predicates which can be assigned to any element of the meta-model. A *package* can include definition of any meta-model element. Packages are mainly used to facilitate the reuse of those definitions.

2.2 UPPAAL Model Checker

In the last decade, the computer science community has made great progress in developing tools and techniques for checking the requirements and design of software systems. One of the most successful approaches is called *model checking*. Figure 2, extracted from [Pal04], shows the main idea behind model checking: the model checking tool takes as an input the requirements or design (called models) and a property (called the specification) that the system should satisfy. The output of the tool is either yes if it satisfies the specification, and no otherwise. [Pal04]

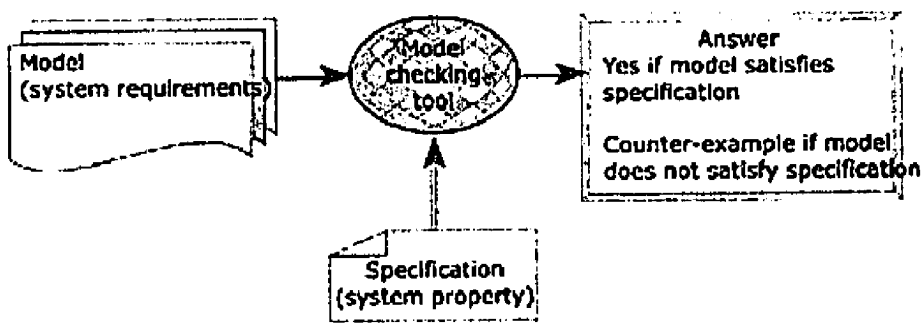

Figure 2: Model Checker Overview

There are many model checking tools in the literature, such as UPPAAL, which is an important example of such tools. UPPAAL is used for the modelling, simulation and

verification of real-time systems, and was jointly developed by Uppsala University and Aalborg University [ABB+01]. It is a mature tool that was developed more than a decade ago, and has been through a number of modifications and improvements. It is designed to verify systems which can be modelled as networks of *timed automata* extended with integer variables, structured data types, and channel synchronization [BDL04].

To be able to understand the UPPAAL model and tool, we need to discuss the following three major concepts: UPPAAL *architecture*, UPPAAL *modeling language* and UPPAAL *ToolKit*.

2.2.1 UPPAAL architecture

Figure 3 shows the architecture of the UPPAAL model. This model represents a system in terms of timed automata (additional details in the next section). In it, a system is defined as a parallel composition of extended timed automata along with global and system declaration sections. An extended timed automata is defined as a *template*. The global declaration section comprises global-level *variables*, *channels*, which are events that can cause synchronous transitions in two timed automata, and user-defined global *functions*. The system declaration part includes instances of templates and system definition as a parallel composition of these instances. Each template contains the definition of a timed automata. The template's major elements are:

- *Local declaration* containing the declaration of local variables and clocks;
- *Locations* defining the states of timed automata. Each state can have an invariant representing a time constraint that should be satisfied in this state;
- *Edges* representing transitions between states. Each edge contains the addresses of two locations, the source and the target of the transition. Each edge can contain:
  - A *guard statement*, which is a constraint that restricts a transition;
  - A *select statement*, which is a statement that assigns values to variables;

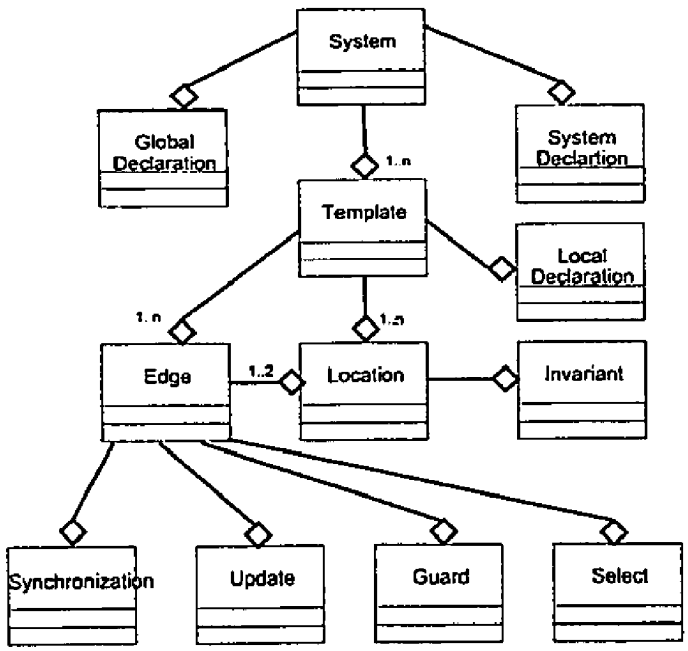

Figure 3: UPPAAL Architecture

- An *update statement*, which represents the post condition of the transition, and assigns values to variables after the transition is performed;
- A *synchronization statement*, which defines the synchronized, shared transition that occurs simultaneously.

2.2.2 UPPAAL modelling language

UPPAAL [BDL04] is designed to verify systems which can be modelled as a network of *timed automata* (TA) that work in parallel. A TA is a finite-state machine extended with clock variables. It can be formally defined as a tuple (L, l_0, K, A, E, I) where:

- L is a set of *locations* denoting the states.
- l_0 is the *initial* state.
- K is a set of *clocks*.
- A is a set of *actions* that cause transitions between locations.

- E is a set of *edges*, $E \subseteq L \times A \times B(K) \times 2^k \times L$, where $B(K)$ is the set of data and time constraints that restrict the transitions and 2^k is the set of clock initializations to set clocks whenever required.

- I is a set of *invariants*, where $I : L \rightarrow B(K)$ is a function that assigns time constraints to clocks.

UPPAAL extends the definition of the TA by defining the following features:

- *Templates*: TA can have *parameters* which are local variables specific to that template, and should be initialized during template declaration.

- *Constants*: These are integer values that cannot be modified and defined as `const name value`.

- *Bounded integer variables*: These are defined as `int[min,max] name`, where `min` is a minimum value and `max` is a maximum value.

- *Binary synchronization*: In UPPAAL, actions causing synchronous transitions are defined as *channels*. A channel can be an *input* (followed by ?) or *output* (followed by !). An edge labelled c! synchronize with another edge labelled c?. Channels are declared as `chan c;`.

- *Broadcast channel*: In broadcast channels, one input action can synchronize with more than one output action. Any receiver can execute its output action, and, if there are none, the input action can still be executed. Broadcast channels are declared as `broadcast chan c;`.

- *Urgent synchronization*: If a channel is defined as urgent, no delays are allowed on the edges containing those channels, and those edges can have no time constraints. They are declared to be `urgent chan c;`

- *Urgent locations*: If a location is defined as urgent, it means that time is not allowed to pass when the system is in this location.

- *Committed locations*: These are more restricted than urgent locations. In a committed location, time cannot pass and also the next transition must contain an outgoing edge of at least one of the committed locations.
- *Arrays*: These can be used with *clocks*, *channels*, *constants* and *integer variables*.
- *Initializers*: These are used to initialize integers and arrays of integers.
- *Expressions*: There are three important types of expression:
  - *Guard*: This is a Boolean expression that can only contain clocks, integers, constants or an array of those types. Guards are used to restrict the behaviour of transitions between locations, and are defined at the edges.
  - *Invariant*: This is a Boolean expression that can only contain clocks, integers, constants or an array of those types. Invariants are defined at the locations, and define the conditions that should always be true.
  - *Assignment*: This expression is used to assign values to clocks and variables at the edges.
- *Edges*: These define transitions between locations, every edge consisting of the following expressions:
  - *Select*, which assigns values to variables within a specific range;
  - *Guard*, which checks that the guard expressions are true;
  - *Synchronization*, which specifies the channels and their directions;
  - *Update*, which resets the values of variables and clocks to the required values.

2.2.3 UPPAAL ToolKit

The main goal of this tool is to model a system with timed automata using a graphical editor, simulate it to validate that the behaviour is as intended and then verify that the model is correct with respect to a certain set of properties. To achieve this goal, the tool was divided into three tabs, as follows:

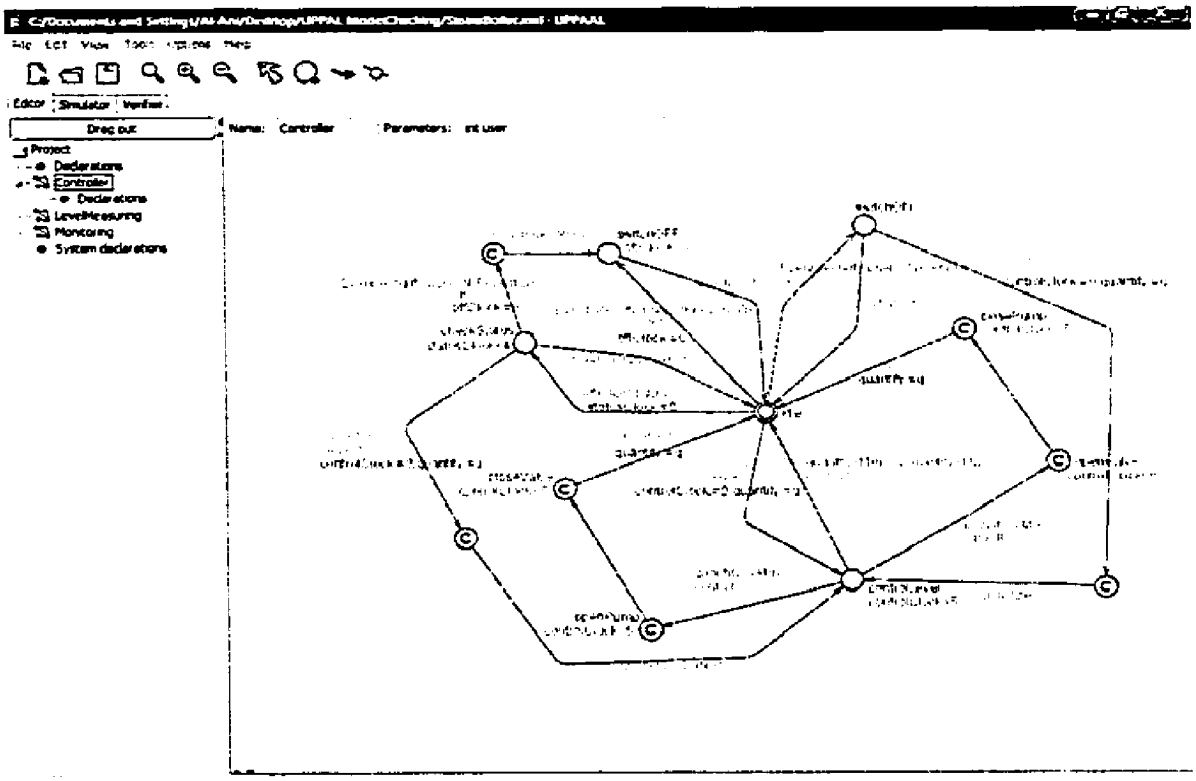

Figure 4: UPPAAL Editor

- *Editor*: Each system consists of TA's (*processes*) that work in parallel. Figure 4 shows the editor part of the UPPAAL tool. The user can use the editor to:

  - Define each *Template* (parameterized TA), by defining the locations, edges, initial locations and properties, and naming each of those elements. The user can also make a *local declaration* that is specific to each template;
  - Define the *Global declarations*, which can be integers, clocks, channels and constants;
  - Define *System declaration*, where templates are instantiated and the contents of the system are defined by assigning instantiated processes to a system.

- *Simulator*: The simulator can trace transactions in three ways: (1) by tracing a predefined set of transactions, (2) by tracing a random set defined by the simulator, or (3) by manually setting transactions chosen by the user, one by one as he goes along.

The main goal of the simulator is to help the user to validate that the model is behaving as expected. It can also catch deadlocks when they occur. Figure 5 shows the simulator part of the UPPAAL tool.

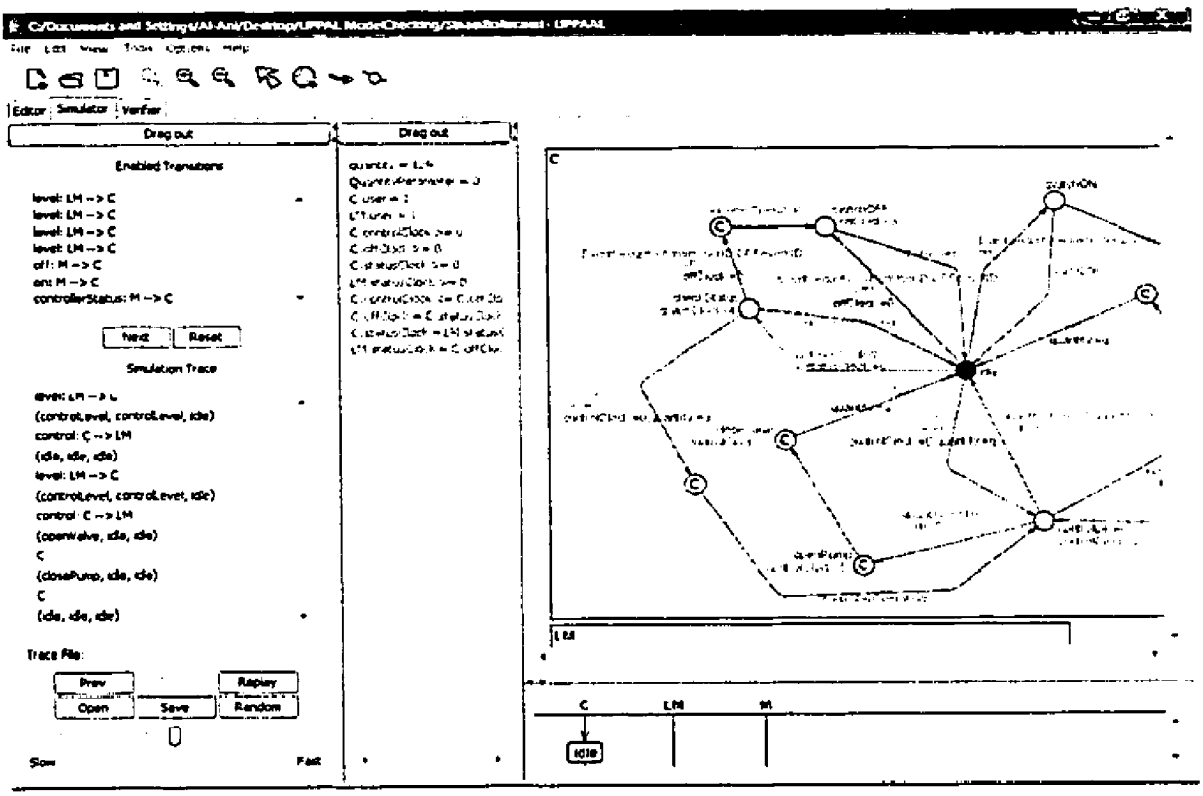

Figure 5: UPPAAL Simulator

- *Verifier*: The user can specify a set of properties to be verified using this feature. Figure 6 shows the verifier part of the UPPAAL tool. In UPPAAL, the checking formula can be a combination of the following [BY04]:
  - A[ ]φ, which means φ *will invariantly happen*
  - E<>φ, which means φ *will possibly happen*
  - A<>φ, which means φ *will always happen eventually*
  - E[ ]φ, which means φ *will potentially always happen*
  - φ--> ψ, which means φ *will always lead to* ψ

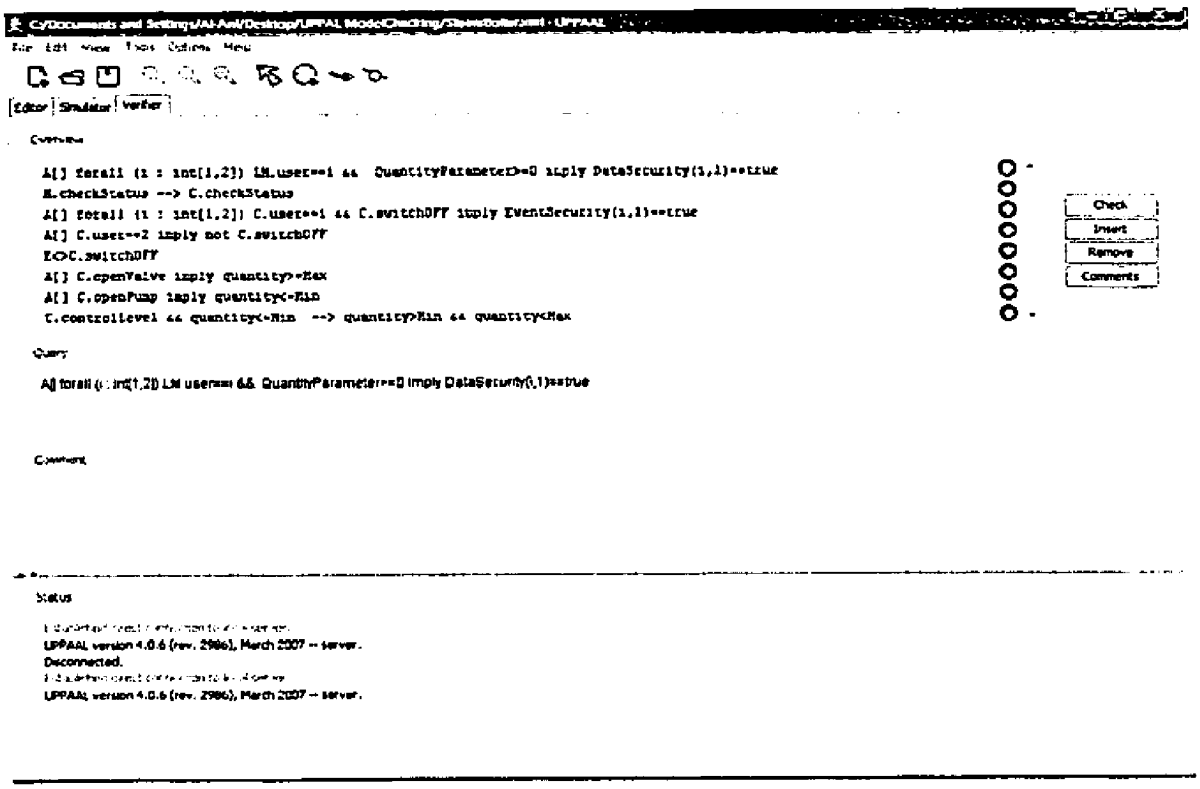

Figure 6: UPPAAL Verifier

Where φ and ψ are Boolean expressions defined on locations, integer variables, and clocks constraints. The properties that can be checked using this verifier are [BDL04]:

- *Reachability:* We can check whether or not it is possible to reach a certain location. We can also check whether or not there is a deadlock in the system, declared in UPPAAL as `A[] not deadlock;`
- *Safety:* We can check whether or not anything bad will ever happen, declared in UPPAAL as something good is always true;
- *Liveness:* We can check whether or not something will happen eventually.

2.3 TIMES Tool

This sections presents a brief introduction of the TIMES tool, a more detailed explanation of which can be found in [AFM+02] and [AFM+03]. TIMES is a modelling and

schedulability analysis tool for embedded real-time systems which was developed in 2001 at Uppsala University. It is suitable for systems which can be described as a set of preemptive or non-preemptive tasks which are triggered periodically or sporadically by time or external events. It provides a graphical interface for editing and simulation, and an engine for schedulability analysis. To be able to understand TIMES and how it works, three major facets of the tool will be discussed: the architecture, the input language and the tool itself (an overview).

2.3.1 TIMES architecture

Figure 7 shows the architecture of the TIMES model. A system defined using it contains the following elements:

- A *global declaration* containing the declaration of the global level variables;
- A *scheduling policy* for the system tasks, which can be: deadline monotonic, rate monotonic, earliest deadline first, first-come first-served, or based on user-defined priorities;
- A *system declaration* containing the instantiation of the templates;
- *Templates*, each representing a TA and containing:
  - a *local declaration* of template-level variables and parameters;
  - *locations*, which represent the state of the TA. Each location may have an *invariant* that defines the time constraint of this location, and can include a task that will be executed at this location;
  - *edges*, which represent the transitions of the TA and may contain synchronization, update, guard and probability statements, all of which expect the probability statement to be the same as in the UPPAAL edges. The probability statement defines the probability that this transition will be chosen.

- *Tasks* define the system-level tasks that can be executed on the template locations. Each can contain multiple attributes, which are: (1) the deadline of the task, (2) the behaviour of the task, which can be sporadic "S", temporarily periodic "TP" or periodic "P", (3) the priority level of the task, (4) the computing time of the task, and (5) the period of the task.

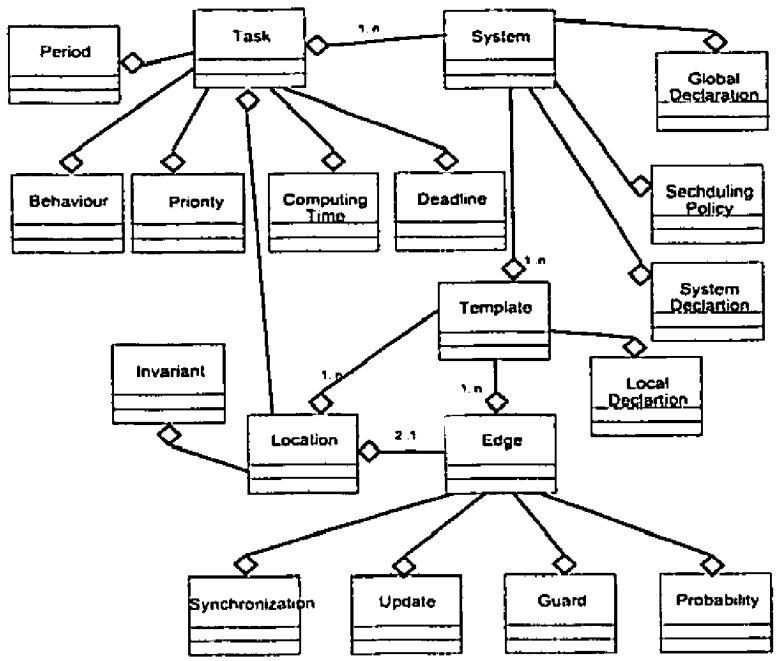

Figure 7: TIMES Architecture

2.3.2 TIMES input language

The heart of the input language of TIMES is the TA with real-time tasks (TAT). A TAT is a TA that has been extended with tasks which are triggered by events. A task is an executable program characterized by its worst execution time and deadline, and possibly other parameters such as scheduling priorities. Each task may have different task instances that are copies of the same program with different inputs. Another major concept in TIMES is that of the task model, which is a task arrival pattern, such as a periodic or sporadic task.

Task Parameters and Constraints

Three types of constraints are addressed in TIMES:

- *Timing Constraints:* A task deadline would be a typical timing contraint;
- *Precedence Constraints:* Task executions may follow specific precedence relations. These relations are described in TIMES by means of a precedence graph in which nodes represent tasks and edges represent precedence relations;
- *Resource Constraints:* These are resources and data variables which are protected by semaphores and may be shared by tasks. A task must follow its semaphore access pattern.

TA as Task Arrival Patterns

As mentioned earlier, the core of the input language is the TA extended with data variables and tasks. As in the UPPAAL model, each edge in the extended TA has labels, which, in TIMES, are the following:

- *guard label*, containing a clock constraint and/or predicate on data variables;
- *action label*, which can be an input or an output action;
- *assignment label*, containing a sequence of assignments in the form x:= 0 or v:= E, where x is a clock, v is a data variable and E is a mathematical expression over data variables and constants.

In the extended TA, a task or set of tasks may be attached to locations which will be triggered when the transition leading to the location is made. The triggered tasks will be placed in a task queue and scheduled to run according to a given scheduling policy. The scheduler is responsible for making sure that all the task constraints are satisfied in scheduling the tasks in the task queue. Networks of automata are constructed, in the standard UPPAAL way, to model concurrency and synchronization between automata.

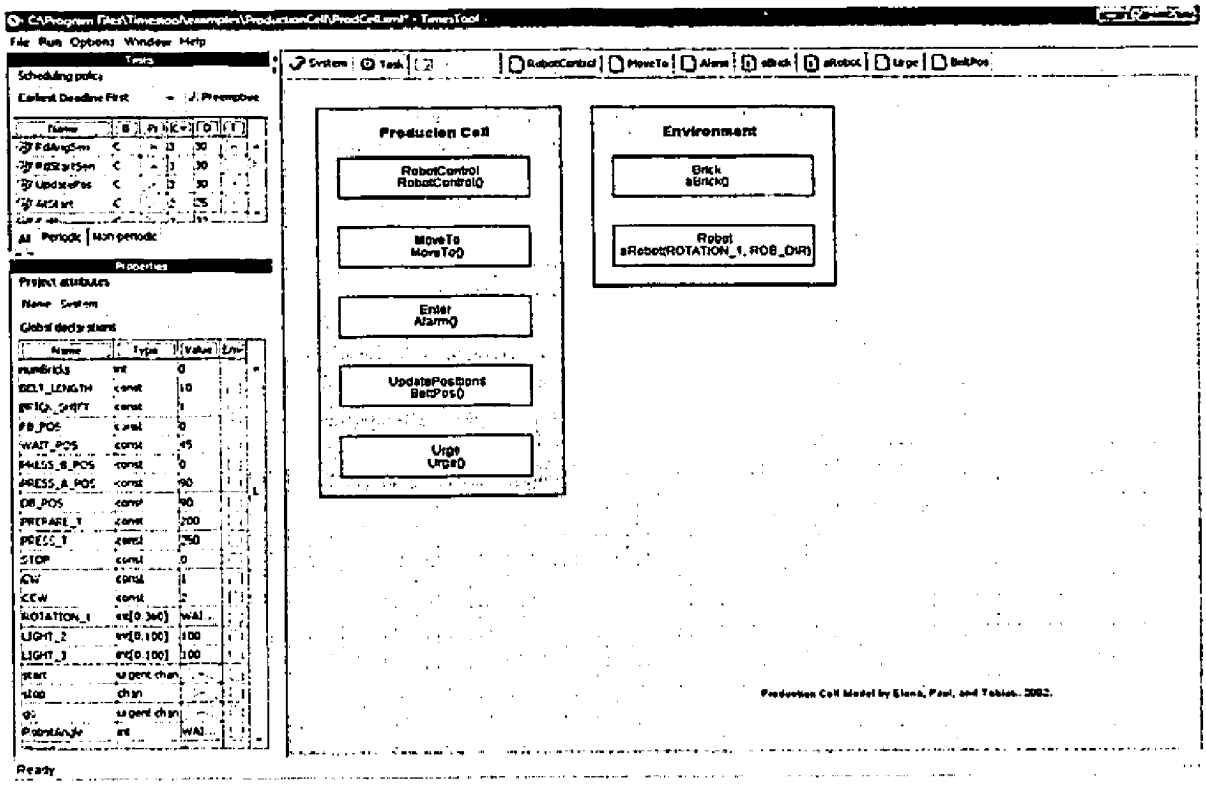

Figure 8: TIMES Tool

Shared Data Variables

There are four ways in which data variables can be used for communication and resource sharing:

- Tasks may share variables with each other.
- Tasks may read and update global variables.
- Automata can read variables owned by the tasks.
- Automata may share variables with each other.

2.3.3 TIMES tool overview

Figure 8 shows a snapshot of the TIMES tool. We begin our discussion of it (an overview) by introducing its main features, which are as follows:

- *Editor:* Which is used to graphically model the abstract behaviour of a system and its environment. The system consists of a network of extended TA with the tasks and a task list. A task is described by: (1) the task code (in C), (2) its worst-case computation time and relative deadline, and (3) optional priority, period and minimal inter-arrival time. Task precedence can be specified using an editor for AND/OR precedence graphs.

- *Simulator:* Which is used to visualize the dynamic behaviour of a system model as Gantt charts and message sequence charts. The simulator can be used in three ways: (1) possible execution traces can be randomly generated, (2) the user can control the execution by selecting the transition to be made, and (3) the error traces produced in the analysis phase can be visualized.

- *Analyzer:* Which is used to check that the tasks associated with a system model are guaranteed to always meet their deadline. If the analysis shows that the task is failing to meet its deadline, a trace is generated and visualized in the simulator. In addition to scheduling, it is possible to analyze the safety and liveness properties specified as temporal logic formulae.

- *Compiler*: : It is responsible for generating executable C code from TA with tasks.

- *Animator*: Which is used to transform hybrid TA modelling the controlled environment into C code simulating the controlled objects in the environment of the embedded system. The simulated environment helps the designer to experiment with the design before implementation.

The TIMES tool performs three functions: system specification, system analysis and code generation.

System specification function

With this function, the user models a system to be analyzed. The function is made up of three parts: (1) the control automata modelled as a networked of extended TA with

tasks, (2) a task table containing information about the processes triggered when the control automata changed locations, and (3) a scheduling policy. The Editor is a tool for drawing the control automata of the system model. It is also used to define the task parameters. The task parameters currently supported are: deadline, execution time, priority, period, a reference to the task code and the task behaviour. The task behaviour can be: sporadic "S", temporarily periodic "TP" or periodic "P". The scheduling policy can be: first-come first-served, fixed priority, rate monotonic, deadline monotonic and earliest deadline first. The Editor output is an XML representation of the control TA. The TIMES tool has a Scheduler Generator. It uses the information from the task table and the scheduling policy to generate a scheduler automaton which is composed in parallel with the controller automata to ensure that the system will behave according the scheduling policy and task parameters.

System Analyzer function

The System Analyzer takes as an input the parallel composition of the control automata and the scheduler automaton. It consists of two main components: a *Simulator* and a *Schedule Analyzer*. The user can explore the dynamic behaviour of the system model and observe how the tasks execute according to the chosen scheduling policy in the Simulator. The Schedule Analyzer performs a schedulability analysis of the system by rephrasing scheduling to a reachability problem that is solved with an extended version of the UPPAAL tool verifier. If the analysis has a negative output, the analyzer generates a trace of the system which ends in a state i.. which one of the system tasks fails to meet its deadline.

Code Generator function

The Code Generator uses the control automata and the task programs to synthesize executable C code. Currently, the only platform supported is the LegOS operating system.

2.4 Summary

This Chapter presented the basic concepts on which the rest of the thesis is built, and has presented, both formally and informally, a summary of the component model definitions. It has also presented the UPPAAL and TIMES tools.

Chapter 3

System Transformation, and the TransformationTool

In this Chapter, the rules and process for transforming a system defined using the TADL model to the UPPAAL or TIMES models are discussed. Section 1 presents the transformation rules and algorithm for the UPPAAL transformation. Section 2 presents the transformation rules and algorithm for the TIMES transformation. Section 3 presents the design of the TransformationTool.

3.1 Transforming the System to the UPPAAL Model

Instead of defining and implementing a new model-checking tool, we can use the UPPAAL model-checking tool, as it is mature and well established. The process of transforming a system defined using the TADL model to the UPPAAL model has been presented in [AM07b]. While that work forms the basis for the work carried out here, the transformation rules defined in [AM07b] have been modified and improved. The main contributions of this work are, then, to improve the transformation rules so that they can be applied to larger-scale models, to define those rules in multiple views to make them easier to understand and, finally, to define the transformation algorithm that will be of major importance in developing the transformation tool.

3.1.1 Transformation rules

Each component-based system is translated to a UPPAAL model in a one-to-one relation, as can be seen in Figure 9. The definition of a component-based system contains a network of connected components and the security mechanism which is based on the role-based security access control (RBAC). The transformation is divided into two parts the *component-level transformation* and the *system-level transformation*.

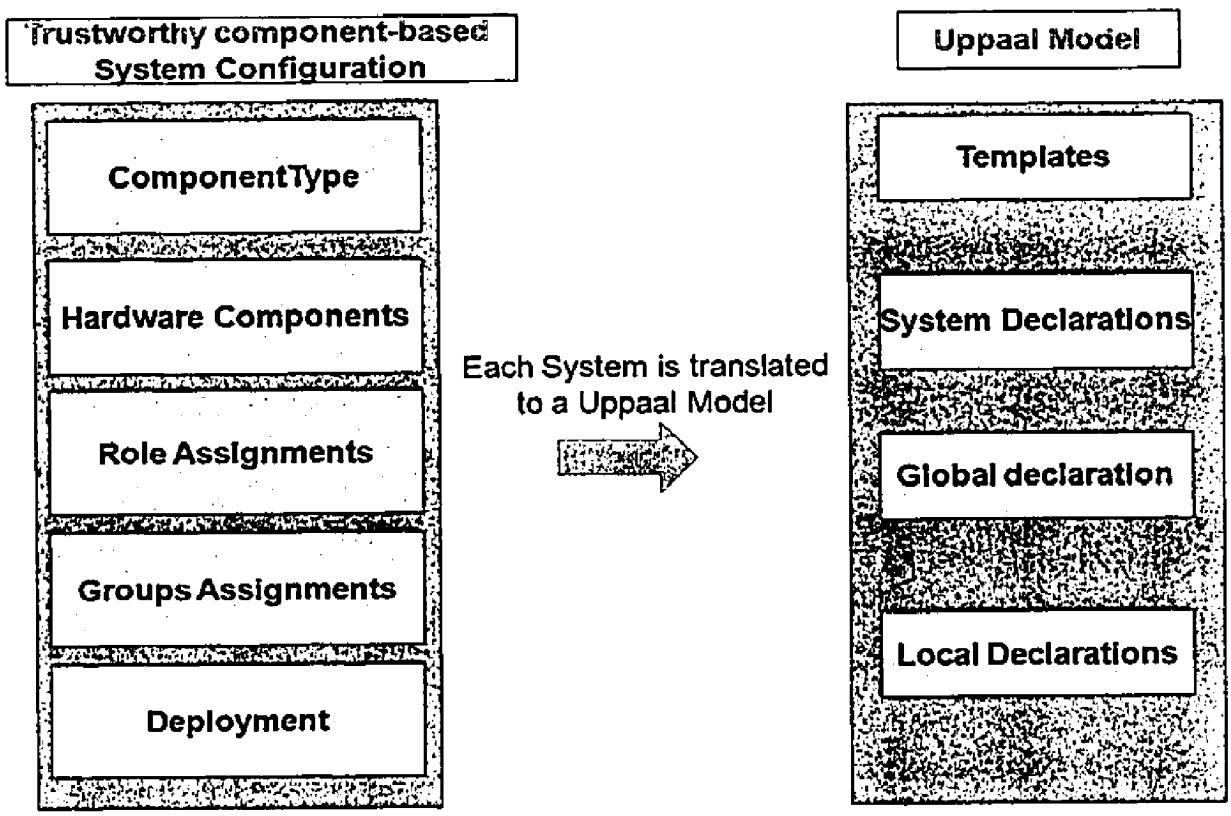

Figure 9: TADL to UPPAAL

Component-level transformation

Each component is translated to a UPPAAL template in a one-to-one relation (Figure 10). As discussed earlier, a UPPAAL template is a TA which can be defined in the tuple (L, l_0, K, A, E, I), where

- L is a list of locations or states;

- l_0 denotes the initial location;
- K is a list of clocks;
- A is a list of actions;
- E is a list of edges;
- I is a function assigning clocks to locations as invariants.

To the tuple we can add u, which is the parameter of the template that represents the user identification (ID) that will be used for security purposes, as will be discussed later on. So the final tuple is (L, l_0, K, A, E, I, u).

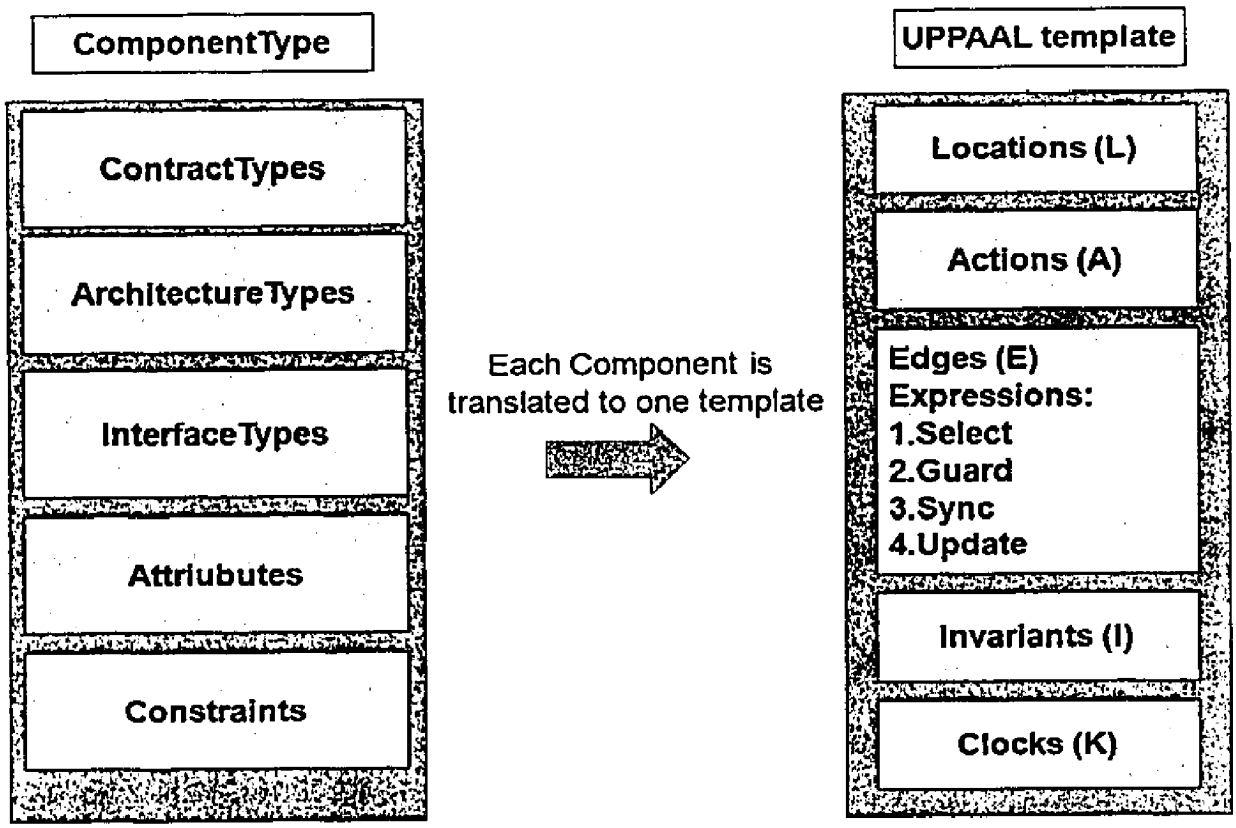

Figure 10: Component to Template

The transformation rules will be described from two points of view, the first in terms of the input units and the second in terms of the output units. The reason for using two different views in the description is to make the transformation rules easier to understand.

Transformation rules in terms of input units

Component: Each component consists of many elements, as shown in Figure 10, one of which is the contract. The contract defines the behaviour of the component, and will therefore be the focus of the transformation.

Contract: Within the contract, we have a list of reactivity rules defining the stimulus-response relationship, where the stimulus represents the event *requesting a service* and the response is the event *responding to a service*. The contents of the contract can be seen in Figure 11.

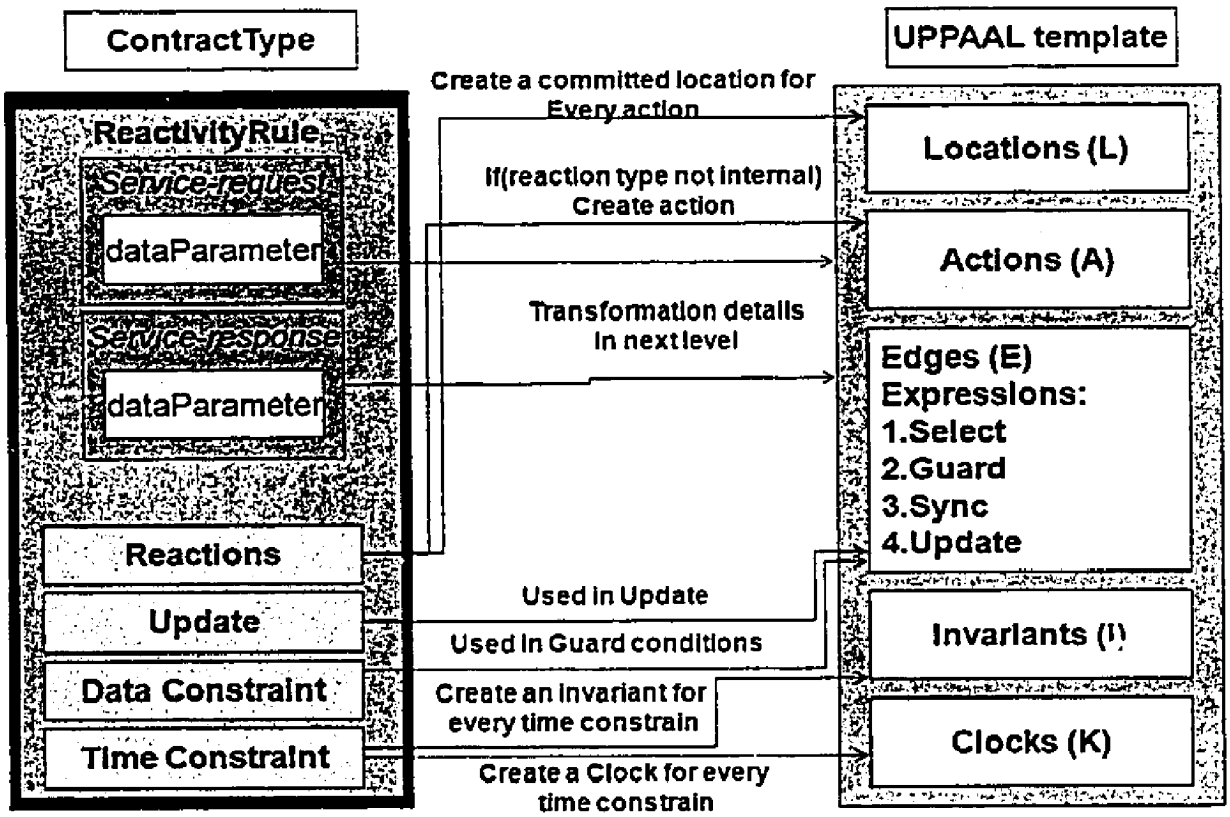

Figure 11: Contract Transformation

Reactivity: Each reactivity has two services, a request service and a response service, the request service representing the stimulus event and the response service representing the response event. As part of the contract, there is also a time constraint and a data constraint, which are used for safety purposes, reactions which define the list of reactions to the stimulus event and the update which defines the post condition of the reactivity. The

transformation rules for the reactivity level are (Figure 11):

- Create a UPPAAL committed location for the list of all reactions.
- Create a UPPAAL action for every reaction that is not internal.
- Create a UPPAAL clock for every time constraint.
- Create a UPPAAL invariant for every time constraint.
- Use time constraint elements in the guard condition on UPPAAL edges.
- Use update elements in UPPAAL update statements on the edges.

Service: Within reactivity, we have the service request and service response events. The transformation rules of those two services are (Figures 12 and 13):

- Create a location for every service request.
- Create an action with the service request name.
- Create an edge from idle to the service request location, and include the created action as a synchronization of the edge.
- Use data parameters for setting values in update statements in the created edges.
- If the Contract Reactions list is not empty, create a committed location for every service response, and create an edge from the service request location to this location.
- If the service request type is not internal, create an action with the service request name.
- If the Contract Reactions list is empty, create an edge back from the service request location to the idle location

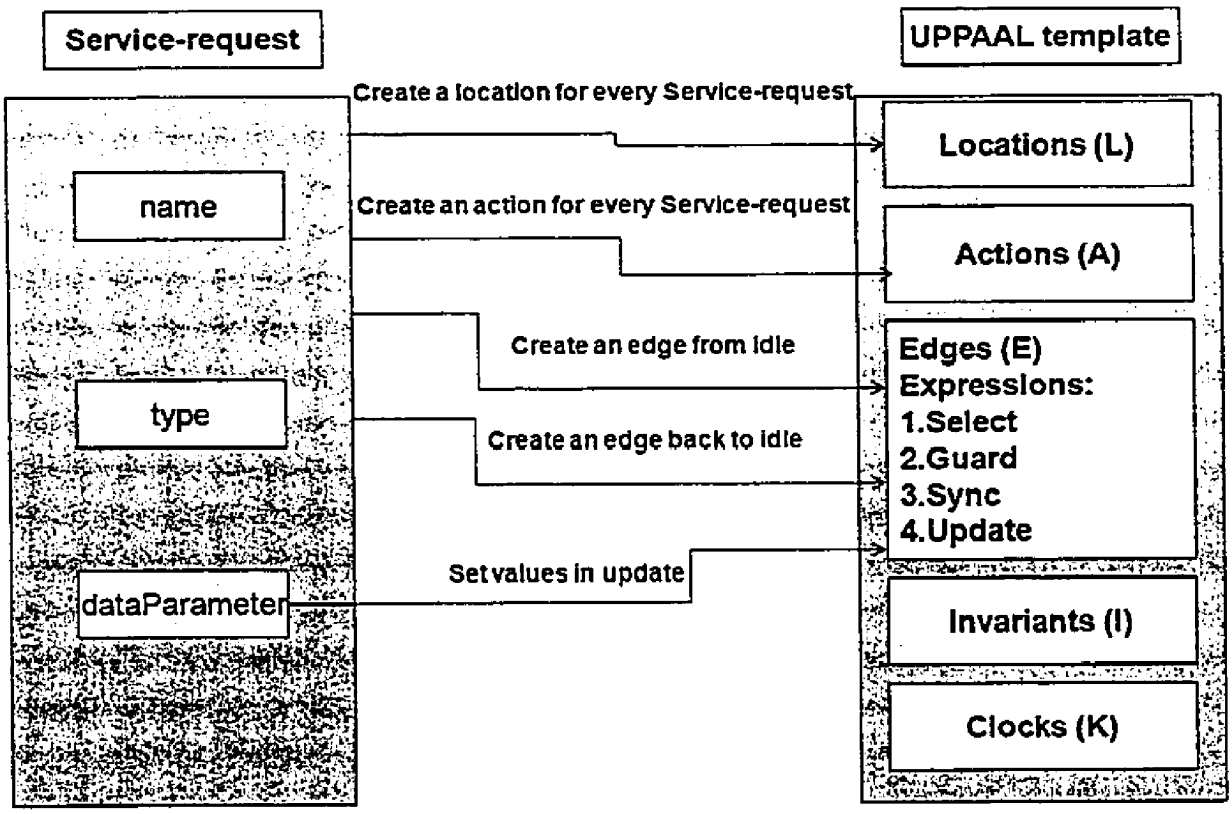

Figure 12: Service Request Transformation

Transformation rules in terms of output units

UPPAAL template: : Each component is translated into one template, and each template has a parameter representing the user ID, which also defines the component. Each template consists of locations, clocks, invariants, actions and edges.

Locations (L): The transformation rules for creating these are as follows:

- Create an initial location l_0 for every template to denote the idle state where the component is waiting for a stimulus.
- Create a location for every stimulus event to represent the processing of this service.
- Create a committed location for every stimulus reaction that is in the reaction list except the last one, as defined in the contract.

Clocks (K): There is only one transformation rule for creating these, which is to create one for every time constraint defined in the contract. Clocks are defined as local variables

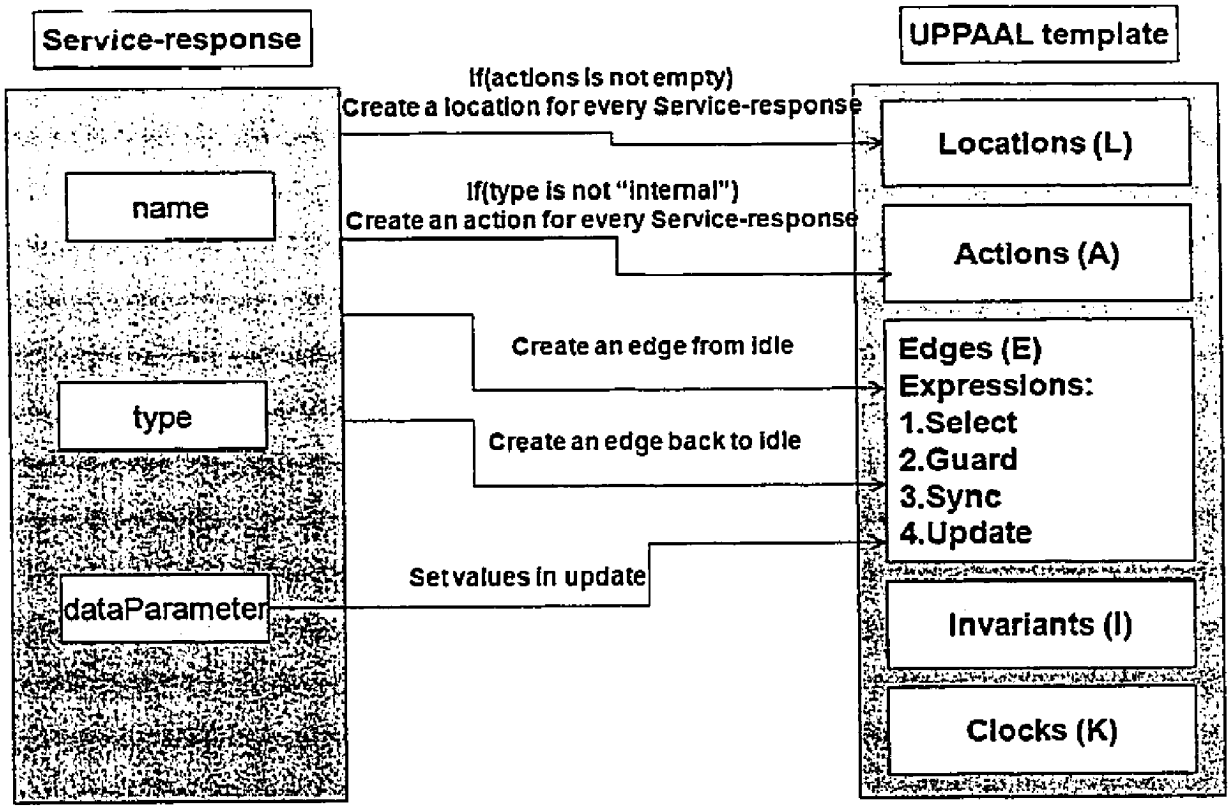

Figure 13: Contract Transformation

with respect to the UPPAAL template.

Invariants (I): The only transformation rule for these is to create an invariant associated with the service request location for every time constraint defined in the contract.

Actions (A): The transformation rules for these are as follows:

- Create an action for every service request.
- Create an action for every service response that is not internal.
- Create an action for every reaction service that is not internal.

Edges (E): As discussed earlier, each edge consists of select, guard, synchronization

and update statements. The select statement contains a temporary variable which is assigned to data parameters in the update statement. The guard statement contains the preconditions that must be met to go through this edge. The synchronization statement contains the actions associated with this edge. The update statement contains the post conditions of the edge. The transformation rules for the edges are:

- Create an edge for every stimulus event from the idle location to the service request location. If there is a time constraint, reset the clock in the update statement.
- Create an edge for every response event, back to the idle location. In the update statement, include the contents of the update element in the reactivity.
- Create an edge for every reaction defined in the reactivity.

System-level transformation

The system definition includes instances of the declared templates. The global declaration section includes the security mechanism. The RBAC defines the security properties of the system in the TADL model. Those properties are transformed to the UPPAAL model using the following rules:

- Create an array of users representing the user IDs that are obtained from the RBAC user list.
- Create an event access control matrix which defines the tuples (user, event, privilege).
- Create a data access control matrix which defines the tuples (user, data, privilege).
- Define an event security function, *EventSecurity*, which searches the event access control matrix of users-events and returns whether or not a user has the privilege of accessing the event.
- Define a data security function, *DataSecurity*, which searches the data access control matrix of data-events and returns whether or not a user has the privilege of accessing the data parameter.

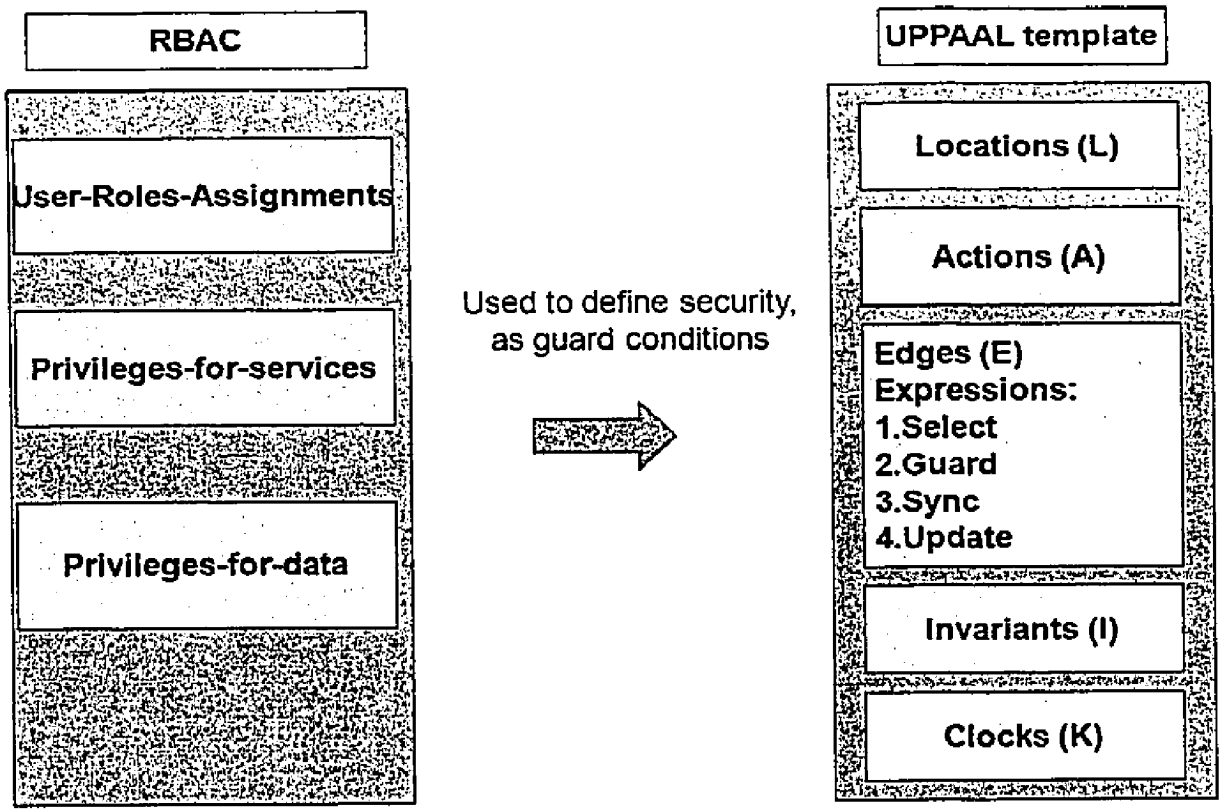

Figure 14: RBAC Transformation

3.1.2 Transformation algorithm

Below is the transformation algorithm that takes as an input the component-based system and returns as an output the UPPAAL model representation of this system.

TADL to UPPAAL transformation algorithm

INPUT : The sets: components CM and System RBAC.

OUTPUT: The sets: locations "L", Edges "E", Clocks "K", Local declaration "LD", Global declaration "GD" and Invariants "I".

From RBAC Create serviceSecurity Matrix and add to GD;

From RBAC Create dataSecurity Matrix and add to GD;

for $Component \in CM$ **do**

Create a new template with name Component.name;

Add Component.parameters as parameters to the new template;

Create a location "idle" to denote the idle state;

R = set of Reactivity inside the contract inside the component C;

for *Reactivity* $\in$ *R* **do**

Create a location "l" called Reactivity.service-request;

Add "l" to "L";

Create an edge "e" from idle to "l" and:

Add Reactivity.service-request in Syn followed by (?) and add channel declaration to GD;

if Reactivity.service-request.parameter =! Empty **then**

for *parameter* $\in$ *Parameters* **do**

Add parameter to Select;

end for

end if

if Reactivity.timeconstrain =! Empty **then**

Create clock "k" and reset it;

Add "k" to "K";

Add "clock k;" to LD;

Add timeConstraint as invariant "q" in location "l";

Add "q" to "I"

end if

Add "e" to "E";

if Reactivity.actions == Empty **then**

Create an Edge "e1" from "1" to idle;

if Reactivity.dataConstraint =! Empty **then**

Add Reactivity.dataConstraint.Constraint to guard;

end if

if Reactivity.service-response.type =! "Internal" **then**

Add Reactivity.service-response in Syn followed by (!) and add channel declaration to GD;

end if

if Reactivity.update =! Empty **then**

Add Reactivity.update to Update;

end if

Add "e1" to "E";

end if

if Reactivity.actions =! Empty **then**

Create a committed location "l1" called Reactivity.service-response;

Add "l1" to "L";

Create an Edge "e1" from "l" to "l1" and:

if Reactivity.dataConstraint =! Empty **then**

Add Reactivity.dataConstraint.Constraint to guard;

end if

if Reactivity.service-response.type =! "Internal" **then**

Add Reactivity.service-response in Syn followed by (!) and add channel declaration to GD;

end if

if Reactivity.update =! Empty **then**

Add Reactivity.update to Update;

end if

Add "e1" to "E";

for $action \in Reactivity.action$ **do**

if not last action **then**

Create a committed location "lx";

```
            Add "lx" to "L";
            Create an edge "ex" from previous action to this action location and:
            if Reactivity.service-response.type =! "Internal" then
              Add Reactivity.service-response in Syn followed by (!) and add channel
              declaration to GD;
            end if
            Add "ex" to "E";
          else
            Create an edge "ex" from previous action location to idle and:
            if Reactivity.service-response.type =! "Internal" then
              Add Reactivity.service-response in Syn followed by (!) and add channel
              declaration to GD;
            end if
            Add "ex" to "E";
          end if
        end for
      end if
    end for
  end for
```

3.2 Transforming the System to the TIMES Model

Transforming a system defined using the component model to the TIMES model is very similar to the transformation to the UPPAAL model. The only major deference is that the TIMES model supports tasks which are not available in the UPPAAL model. Instead of repeating the transformation rules defined in the previous section, only the differences will be discussed here.

3.2.1 Transformation rules

Each component-based system is also translated into one TIMES system, and each component is translated into one template. The main differences in the transformation rules can be summarized as follows:

- In TIMES, we need to define the tasks, and the transformation rule for this is: create a task for every service in the model, map the service attributes into task attributes.
- Some templates in TIMES are environmental templates, and whether or not a component is translated into an environmental or regular template is decided by an attribute defined in the component.
- In TIMES transformation, we need to assign tasks to locations. The transformation rule for this is: for every location created, add to this location a task that represents the service proceeding at this location.
- In UPPAAL, we had an RBAC transformation, which is not supported in TIMES. The analysis performed in TIMES is concerned with the schedulability analysis, not the safety and security analysis.
- In TIMES, edges do not contain *select* statements, and so the *select* statements defined in the reactivities will be ignored.

3.2.2 Transformation algorithm

The transformation algorithm is very similar to the UPPAAL transformation algorithm. The major difference is in the definition of tasks. So, below is an algorithm for creating those tasks, which will be added to the locations representing the processing of the service response.

TADL to TIMES transformation algorithm

INPUT : The sets: components CM and System RBAC.

OUTPUT: The set of Tasks "T".

for *Component* $\in CM$ **do**

R = set of Reactivity inside the contract inside the component C;

for *Reactivity* $\in R$ **do**

Create a task with name Reactivity.service-response;

P = set of properties inside the Service-response;

for *Property* $\in P$ **do**

Create a property in the task with the same name and value;

end for

if Reactivity.timeConstraint=! Empty **then**

Create a property in the task with the name "D" and value of the timeConstriant ;

end if

end for

end for

3.3 Transformation Tool

As part of the work done in this thesis, a tool was developed to automate the transformation from a real-time reactive system developed using the trustworthy component-based model (TADL model) to an extended TA understood by UPPAAL or an extended TA understood by TIMES. This tool is called the TransformationTool. Figure 15 shows the process for transforming the model to the UPPAAL model. A similar flow chart can be used to describe the transformation to the TIMES model. The first element is the TADL XML file, which represents the input to the transformation tool. The tool can then perform two types of transformation, depending on the transformation selected. It can result in two types of XML files. If the UPPAAL transformation is selected, the XML file produced adheres to the UPPAAL model, and the resulting file is input to the UPPAAL model checking tool to perform the required verification and simulation, as can be seen in Figure 15. If

TIMES transformation is selected, the XML file produced adheres to the TIMES model, and the resulting file is input to the TIMES tool, where schedulability analysis can be performed. The following subsections will introduce the architecture and design of the TransformationTool.

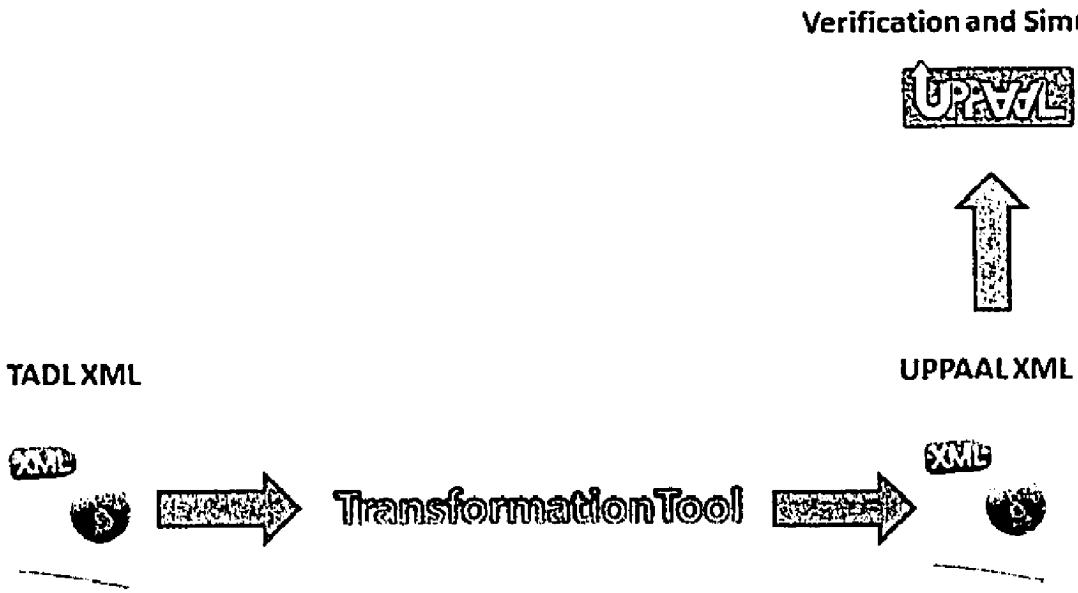

Figure 15: Transformation Flow Chart

3.3.1 Architecture overview

The software architecture defines the hierarchical structure of the program components, and the way these components interact with one another and the structures of data that are used between components [Pre01]. In [SG96], software architecture is defined as the structure or structures of the system, which includes software components, the externally visible properties of those components and the relationships between them. There are many software architecture styles, such as the *client-server system*, *event-driven*, *peer-to-peer* architectures, and the *pipe and filter* architecture. The choice was made to use the pipes and filters architecture in the design of the Transformation Tool, and below is an introduction to this architecture, its advantages and the rationale behind this choice.

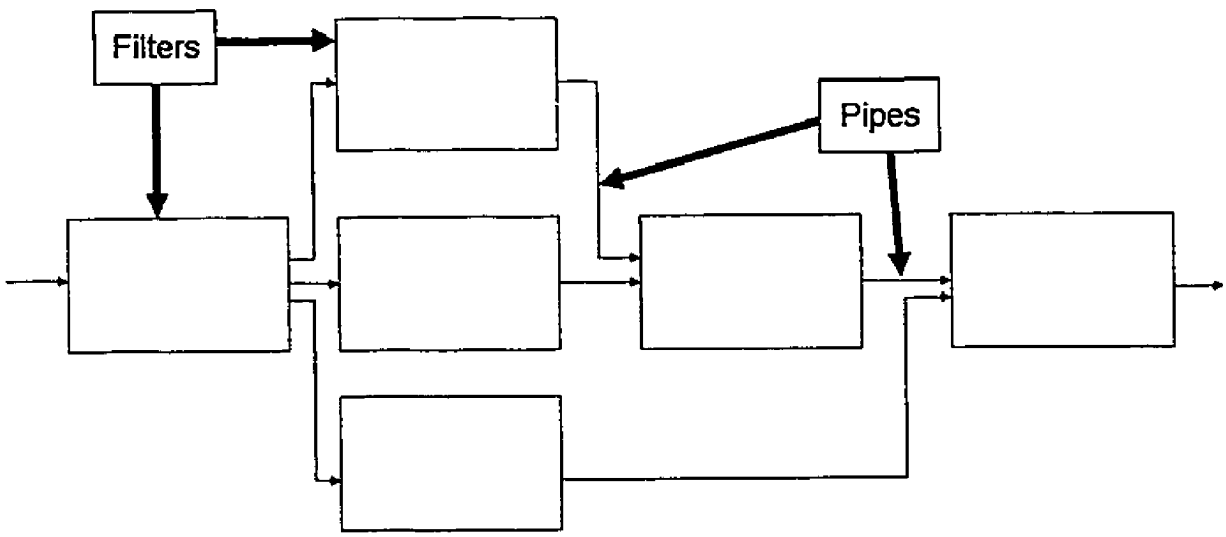

Figure 16: Pipe and Filter Architecture

Pipe and Filter architecture

In the Pipe and Filter style, each component has a set of inputs and a set of outputs. A component reads a stream of data as input and generates a stream of data as output. This process usually includes data transformation, which makes the components act as filters, and hence they are called *filters*. By contrast, the connectors act as conductors for the data streams from one filter to another, which makes the connectors act as pipes, and hence they are called *pipes*, and the whole architecture is called *Pipe and Filter*. [GS94]. Pipes and filters are usually connected sequentially, where the input to the first filter is the input to the system, and the output of the last filter is the output of the system. Figure 16 shows a simple view of this architecture, which has a number of advantages:

- It makes the system behaviour easier to understand on the part of the designer. [GS94]
- It supports the reusability of the filters. [GS94]
- It increases the maintainability of the system; filters can be maintained separately, and they can be replaced by other filters with better performance. [GS94]
- It ensures low coupling, in that filters only interact in a limited way. [LBK98]

- It eases the concurrent executions of the system, where filters can be run on multiple processors or multiple threads on the same processor. [LBK98]

Rationale behind the selection

Pipe and Filter is the architectural style we used in our design because it makes the system behaviour easier to understand, and it improves reusability and maintainability. As this is a research tool, it is very important that the design be clear for future users who might need to improve it. It is also important to increase its reusability, as some components may be required for other tools. Finally, maintainability is an essential property for a research tool, as it is very likely that more improvements will be made to it in the future.

3.3.2 Architecture diagram

Figure 17 represents the *component diagram* of the TransformationTool, and shows the components that make up the tool. As the figure shows, there are five components which communicate with one another following the Pipe and Filter architecture through the pipes. These are as follows:

- The first component is the *Transformation Rules* component, which defines the rules for transforming the TADL model to the output model. The output model can be the UPPAAL model or the TIMES model. The rationale behind defining the transformation rules in a separate component is to increase the maintainability and extensibility of the tools. If a modification to the transformation rules is needed, all that is required is to modify the transformation rules component, and there is no need to go into the tool implementation details.

- The second component is the *TADL XML* component, which is the XML file containing the definition of the system which satisfies the trustworthy component-based model XML schemas.

- The third component is the *Transformation Process and graphical user interface*

(GUI) component. This component will be responsible for performing the transformation on the TADL XML file following the transformation rules defined in the transformation rules component.

- The fourth component is the output of the transformation generated from the previous component. This output can either be a UPPAAL XML file or a TIMES XML file, depending on the type of transformation performed.

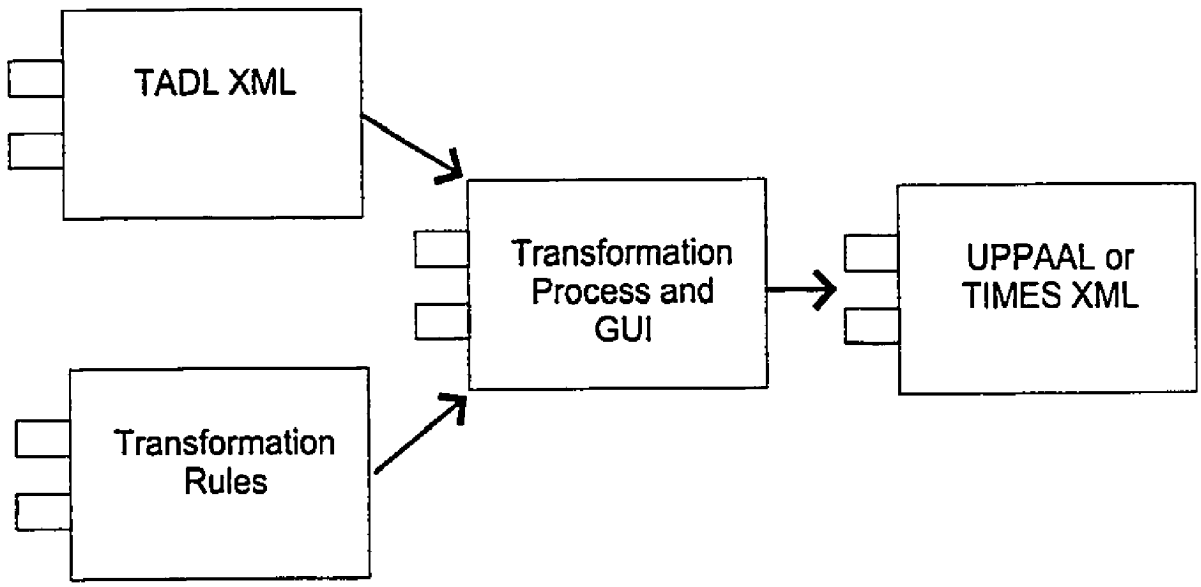

Figure 17: Component Diagram

3.4 Summary

In this Chapter, the transformation process, rules and algorithm for transforming a TADL system to the UPPAAL or TIMES models were presented. The TransformationTool was introduced and the architectural design of this tool was discussed.

Chapter 4

Transformation Tool Implementation

In this Chapter, the implemnation of the TransformationTool is discussed. As explained in Chapter 3, the design of the TransformationTool contains four major components. In the following sections, the implementation of each of those components will be described, and a demonstration of the way the TransfromationTool works will be presented.

4.1 Transformation Rules Component

This component incorporates the rules for transforming a system defined in the TADL model to the UPPAAL model or the TIMES model. These rules, which were discussed in Chapter 3, were defined separately in a single component to increase the tool's extensibility and maintainability by ensuring that a change in the transformation rules will not affect the implementation of the rest of the tool. There are many ways to represents these transformation rules, using standard programming languages such as Java, or specific XML languages such as XSLT. We chose to use XSLT. Below is an introduction to XSLT, its advantages and our rationale for choosing it.

4.1.1 XSLT

Extensible Stylesheet Language Transformations (XSLT) is a powerful and flexible XML-based language for transforming XML documents into other formats, for example an HTML

document, another XML document, a Portable Document Format (PDF) file, a Scalable Vector Graphics (SVG) file, a Virtual Reality Modeling Language (VRML) file, Java code, a flat text file or a JPEG file, among many others [Tid01]. XSLT was developed by the World Wide Web Consortium (W3C). The most recent version is XSLT 2.0, which reached W3C recommendation status on January 23, 2007, and will be used here. Figure 18 shows the XSLT processing model, which includes the following:

- one or more XML source documents;
- one or more XSLT stylesheet modules;
- the XSLT template processing engine (the processor);
- one or more result documents.

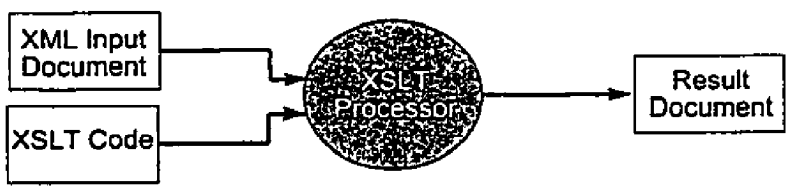

Figure 18: XSLT Processing Model

The dominate feature of a typical XSLT stylesheet (XSLT code) is that is consists of a sequence of template rules, each of which describes how a particular element type or other construct should be processed. The rules are not arranged in any particular order, they don't have to match the order of the input or the order of the output. This is what makes XSLT a declarative language, because you specify what output should be produced when particular patterns occur in the input, while in procedural language you have to say what tasks to perform in what order.

Following is a simple example of how to use XSLT to change the structure of an XML file. This is done by transforming the original XML file to a new file that have the new structure. In our example, the input XML file have a list of students, each student is defined by its name and address. The name is divided into first and last. The address is divided into

```
<student>
   <name>
      <first>Mike </first>
      <last>Jack </last>
   </name>
   <address>
      <unit>4 </unit>
      <street>King St, </street>
      <city> Ottawa, </city>
      <country> Canada </country>
   </address>
</student>
```

Figure 19: Input XML file

street, city, province and country. In the output XML file we need the name to be presented as a one XML element and the same for the address. Figure 19 shows the input XML. Figure 20 shows the XSLT stylesheet that defines the transformation rules. It specifies that for each `<name>` tag in the input file create a new XML tag in the output file called `<FullName>` it contains the string associated with the two XML tags `<first>` and `<last>` defined in the input file. And for every `<address>` tag in the input file create a new XML tag in the output file called `<FullAddress>` that contains the strings of the `<street>`, `<city>`, `<province>` and `<country>` tags. The resulted XML file can be seen in Figure 21.

XSLT has many advantages, among them the following:

- It is specifically designed for XML, which means that there is no need to worry about the details of reading or writing XML files. This is not the case with other languages like Java.
- The transformation code is smaller than other languages, which makes it easier to understand and improves its maintainability.
- It enables us to define the transformation rules and process at the same time.

```
<xsl:stylesheet xmlns:xsl="'http://www.w3.org/1999/XSL
   /Trasnform" version="2.0">

   <xsl:output method="xml"/>

   <xsl:for-each select="name">
      <FullName>
         <xsl:sequence select="xs:string(name/first)"/>
         <xsl:sequence select="xs:string(name/last)"/>
      </FullName>
   </xsl:for-each>

   <xsl:for-each select="address">
      <FullAddress>
         <xsl:sequence select="xs:string(address/unit)"/>
         <xsl:sequence select="xs:string(address/street)"/>
         <xsl:sequence select="xs:string(address/city)"/>
         <xsl:sequence select="xs:string(address/province)"/>
         <xsl:sequence select="xs:string(address/country)"/>
      </FullAddress>
   </xsl:for-each>

</xsl:stylesheet>
```

Figure 20: XSLT stylesheet

```
<student>
   <FullName>Mike Jack</FullName>
   <FullAddress>4 King St, Ottawa, Canada </FullAddress>
</student>
```

Figure 21: Output XML file

- XSLT stylesheets are XML documents, which makes it feasible to manipulate them programmatically to produce new stylesheets on the fly.
- The tools and support are widely available in the industry. XSLT is an open standard, and there are several open-source XSLT engines available for Java, such as Xalan and Saxon.

4.1.2 Rationale behind selection

XSLT was chosen to represent the transformation rules for many reasons, the most important, besides the advantages discussed earlier, being the ability to formally define the transformation rules separately. Component models evolve with time. New concepts can be added, and existing concepts can be removed or updated. Therefore, the transformation rules may require continuous updates which can be easily achieved using XSLT. Also, it is possible for the transformation process to use many different types of sources. Therefore, using XSLT enables easy management and evolution of the transformation process with no need to redevelop, rebuild or redistribute the model's Transformation Tool.

4.1.3 Transformation rules in XSLT

Each transformation rule presented in Chapter 3 should be formally defined in XSLT, and each set of transformation rules implemented in XSLT in a single file. The UPPAAL transformation is defined in one file, and so is the TIMES transformation. The full contents of those two files can be found in Appendix B. Below is a detailed discussion of the UPPAAL and TIMES transformation rules in XSLT.

UPPAAL transformation rules

The rules are defined in a single file, the structure of which is as follows. First, it includes definitions for creating the global declaration section. Then, it defines the rules for transforming each component into one single template. The template definition contains the

rules for transforming the locations and edges, and the local-level declarations. Finally, it defines the system-level declaration. Below is a more detailed discussion.

Global declaration rules

The global declaration is defined in UPPAAL as an XML tag `<declaration>` containing a string. The global declaration rules are responsible for generating the global declaration for the UPPAAL system. This declaration will contain:

- the global-level variables, which are transformed from the TADL global level attributes;
- the channel declaration, which represent the synchronization events that trigger the transitions;
- the user ID declaration (each user identity is given a unique integer value);
- the definition of the security mechanism, which includes:
  - a User Event Access Matrix defining the user's right of access to each event;
  - a User Data Access Matrix defining the user's right of access to each data parameter;
  - the Event Security function, which retrieves the correct right of access from the User Event Access Matrix and the Data Security function, which in turn retrieves the correct right of access from the User Data Access Matrix.

Figure 22 shows a simple portion of the transformation rules for declaring the channels in the global declaration. The rule states that, *"for every service-request (stimulus) in every reactivity, create a channel in the UPPAAL template."* This example shows how we utilized the XSLT function *"for-each-group"* to loop through all the reacvtivities and perform the required transformation. *"for-each-group"* was used instead of *"for-each"* because it sorts the result into groups, depending on the values defined in *"group-by"*. This sorting is important to enable us to overcome the problem of repetition, as we can sort the entries

```
...
<declaration>
....
   <xsl:for-each-group select="components/contract/reactivity/
         service-request" group-by="name">
      <xsl:variable name="p" select="position()"/>
      <xsl:for-each select="current-group()[1]/name">
         <xsl:sequence select="fn:concat('chan ', xs:string(.),
         '; const int ',xs:string(.),'ID = ',$p,' ;',$newline)"/>
      </xsl:for-each>
   </xsl:for-each-group>
...
</declaration>
......
```

Figure 22: Sample Global Declaration Transformation Rule

and choose only the first element and perform the transformation on that element. As can be seen in the previous example, if we had used "*for-each*", we would have a multiple-channel declaration for the same service request, as it may be a member of more than one reactivity. XPath expressions were used to locate the source items in the TADL XML file. For example, *select= "contract/reactivity/service-request"* is used to locate all the service requests in reactivity definitions. Then, the XPath function *"fn: concat(String, String)"* is used to declare the channel name, which is the same as the stimulus name. Finally, the resulting string is added to the global declaration.

Template transformation rules

Every template corresponds to one component. Each template consists of: name, list of parameters, local declaration, locations and edges. The name of the template is the same as the name of the corresponding component. The list of parameters contains the user identity. The local declaration contains the declaration of the local-level variables and clocks declaration. Figure 23 shows the transfromation rules for the template name, list of parameters and local declaration. The transformation rule for declaring clock states that: *"for every time constraint create a clock with the time constriant name."* The XPath

```
...
   <xsl:for-each select="components">
      <template>
         <xsl:for-each select="name">
            <name>
               <xsl:sequence select="xs:string(.)"/>
            </name>
         </xsl:for-each>
         <parameter>int user</parameter>
         <declaration>
            <xsl:for-each-group select="contract/reactivity/
               timeConstraint"group-by="name">
               <xsl:for-each select="current-group()[1]/name">
                  <xsl:sequence select="fn:concat('clock ',
                     xs:string(.),';')"/>
               </xsl:for-each>
            </xsl:for-each-group>
            <xsl:for-each select="attribute">
               <xsl:variable name="name" select="name"/>
               <xsl:variable name="type" select="datatype"/>
               <xsl:variable name="value" select="value"/>
               <xsl:sequence select="fn:concat($type,' ',$name,
               ' = ',$value,' ;')"/>
               </xsl:for-each>
         </declaration>
......
```

Figure 23: Sample Template Declaration Transformation Rule

function *sequence* is used through the transformation rules to write strings inside the XML tags. XSLT variables were used to hold temporary values inside the local block. One important note on the use of XSLT variables is that the value of those variables cannot be changed, and they are only defined inside the block that contains them. For instance, in the previous example, the variable "name" is only defined within the scope of the *"for each"* loop

The location transformation rules were discussed in Chapter 3. Figure 24 shows a simple portion of the transformation rules for creating UPPAAL locations. This rule states

```
...
<xsl:for-each-group select="contract/reactivity/service-request"
      group-by="name">
   <xsl:variable name="nuuu" select="count(current-group())"/>
   <xsl:for-each select="current-group()[1]/name">
      <location>
         <xsl:attribute name="id">
            <xsl:sequence select="xs:string(.)"/>
         </xsl:attribute>
         <name>
            <xsl:sequence select="fn:concat(xs:string(.),'S')"/>
         </name>
         <xsl:variable name="vv" select="../../timeConstraint
            /name"/>
         <xsl:variable name="max" select="../../timeConstraint/
            maxSafeTime"/>
         <xsl:if test="../../timeConstraint">
            <label kind="invariant">
               <xsl:sequence select="fn:concat($vv,' &lt; '
                  ,$max)"/>
            </label>
         </xsl:if>
         <xsl:if test="$nuuu = 1">
            <xsl:if test="../../service-response/type='internal'">
               <committed/>
            </xsl:if>
            <xsl:if test="../../service-request/type='internal'">
               <committed/>
            </xsl:if>
         </xsl:if>
      </location>
   </xsl:for-each>
</xsl:for-each-group>
...
```

Figure 24: Sample Location Transformation Rule

that, *"for every service-request (stimulus) in every reactivity, create a location in the UPPAAL template. If this reactivity has a time constraint, then add an invariant to the location. Also, if either the stimulus or the response is an internal event, then make the location committed (a committed location is a location where time does not pass)."* We use XPath expressions to locate the source items in the TADL XML file. For example, *select="contract/reactivity/service-request"* is used to locate all the service requests in the reactivity definitions. Then, a location with the same name is created as an XML tag `<location>`, which conforms to how UPPAAL defines locations. Later, attributes are added to the location. We used the XSLT function *"xsl:if"* for condition control, which is very similar to *"if statements"* in traditional programming languages. The *xsl:attribute* is used to define XML tag attributes.

The edge transformation rules were discussed in detail in Chapter 3. Figures 25, 26 and 27 show a simple portion of the transformation rules for creating UPPAAL edges. This rule states that, *"for every service-request (stimulus) in every reactivity, create an edge in the UPPAAL template. The edge starts from the idle location and ends at the location representing the processing of the stimulus. If the stimulus is not an internal event, add the channel that represents the stimulus in synchronisation statements on the edge. If the reactivity has a select statement, add this statement to the edge. If the reactivity has a time constraint, assign the clock that has the same name as the time constraint to zero and add it to the assignment statement on the edge."* We use the XPath expressions *"for-each-group"*, *"group-by"* and "count" to ensure that, in case of multiple service-requests which are used in different reactivities, only one edge is created in the UPPAAL template, as discussed earlier. The XML tag `<transition>` represents the edge in the UPPAAL model. The XML `<source>` and `<target>` tags define the start- and end-points of the transition. After defining the source and target of the transition, checks are made in order to decide whether or not to include the synchronization, select, guard and assignment statements following the rule presented above.

The last part of the transformation rules contains the transformation rules for creating the system-level declaration. That declaration is responsible for creating instances of the

```
....
<xsl:for-each-group select="contract/reactivity/service-request"
   group-by="name">
   <xsl:variable name="nu" select="count(current-group())"/>
   <xsl:for-each select="current-group()[1]/name">
      <transition>
         <source>
            <xsl:attribute name="ref">
               <xsl:sequence select="'idle'"/>
            </xsl:attribute>
         </source>
         <target>
            <xsl:attribute name="ref">
               <xsl:sequence select="xs:string(.)"/>
            </xsl:attribute>
         </target>
         <xsl:if test="not(../type = 'internal')">
            <label kind="synchronisation">
               <xsl:for-each select="../../service-request/name">
                  <xsl:sequence select="fn:concat
                     (xs:string(.),'?')"/>
               </xsl:for-each>
            </label>
         </xsl:if>
         <xsl:if test="../type = 'internal'">
            <xsl:if test="$nu = 1">
               <label kind="synchronisation">
                  <xsl:for-each select="../../service-response
                     /name">
                     <xsl:sequence select="fn:concat(xs:string(.),
                        '!')"/>
                  </xsl:for-each>
               </label>
            </xsl:if>
         </xsl:if>
...
```

Figure 25: Sample Edge Transformation Rule Part 1

```
...
<xsl:if test="../../select">
   <xsl:for-each select="../../select">
      <xsl:variable name="na" select="name"/>
      <xsl:variable name="ty" select="type"/>
      <xsl:variable name="min" select="min"/>
      <xsl:variable name="max" select="max"/>
      <xsl:variable name="t" select="to"/>
      <label kind="select">
         <xsl:sequence select="fn:concat($na,':',$ty,
         '[',$min,',',$max,']')"/>
      </label>
     <xsl:if test="../timeConstraint">
        <xsl:variable name="vv" select="../
        timeConstraint/name"/>
        <label kind="assignment">
           <xsl:sequence select="fn:concat($vv,':=0')"/>
           <xsl:sequence select="fn:concat(',',$t,':='
           ,$na)"/>
        </label>
      </xsl:if>
   </xsl:for-each>
</xsl:if>
<xsl:if test="not(exists(../../select))">
   <xsl:if test="../../timeConstraint">
      <xsl:variable name="vv" select="../../
      timeConstraint/name"/>
      <label kind="assignment">
         <xsl:sequence select="fn:concat($vv,':=0')"/>
      </label>
   </xsl:if>
</xsl:if>
...
```

Figure 26: Sample Edge Transformation Rule Part 2

```
....
         <label kind="guard">
            <xsl:if test="not(../type = 'internal')">
               <xsl:sequence select="fn:concat
               ('EventSecurity(user,',xs:string(.) ,'ID)')"/>
               <xsl:for-each select="../parameterType/name">
                  <xsl:variable name="pamName"
                     select="xs:string(.)"/>
                  <xsl:for-each select="/Configuration/rbac/
                  privilegesForDataParameters/dataParameter/name">
                     <xsl:variable name="temp"
                        select="xs:string(.)"/>
                     <xsl:if test="$pamName = $temp">
                        <xsl:sequence select="fn:concat('&amp;&amp;
                        DataSecurity(user,',xs:string(.),'ID)')"/>
                     </xsl:if>
                  </xsl:for-each>
               </xsl:for-each>
            </xsl:if>
            <xsl:if test="$nu = 1">
               <xsl:if test="../../dataConstraint">
                  <xsl:for-each select="../../dataConstraint
                        /constraint">
                     <xsl:sequence select="fn:concat('&amp;&amp; ',
                     xs:string(.))"/>
                  </xsl:for-each>
               </xsl:if>
            </xsl:if>
         </label>
      </transition>
   </xsl:for-each>
</xsl:for-each-group>
...
```

Figure 27: Sample Edge Transformation Rule Part 3

```
<system>
   <xsl:for-each select="components">
      <xsl:sequence select="fn:concat('comp',position(),' = '
         ,xs:string(name),'(adminID);',$newline)"/>
   </xsl:for-each>
   <xsl:sequence select="xs:string('system ')"/>
   <xsl:variable name="c" select="count(components)"/>
   <xsl:for-each select="components">
      <xsl:if test="position() != $c">
         <xsl:sequence select="fn:concat('comp',position(),',')"/>
      </xsl:if>
      <xsl:if test="position() = $c">
         <xsl:sequence select="fn:concat('comp',position(),';')"/>
      </xsl:if>
   </xsl:for-each>
</system>
```

Figure 28: System Declaration Transformation Rule

created templates by passing values for the template parameters. Figure 28 shows the transformation rules for creating the system-level declaration. The rule states that, *"for each component that has been translated to a template, create one template instance and pass the component user as the template parameter."*

TIMES transformation rules

TIMES transformation rules are also defined in a single XSLT file. The transformation rule implementation is very similar to that of the UPPAAL transformation rules. The main difference is the way TIMES defines its elements, as it follows a different XML schema than UPPAAL. But the same concept is applied as in UPPAAL. TIMES has an extra concept, which is the task concept. Figure 29 shows the transformation rules for creating those tasks in TIMES. The transformation rule states that, *"for every service request (stimulus), create a task that represents the processing of this event."* The full transformation rules can be found in Appendix B.

```
....
<tasktable schedulingpolicy="EDF">
   <xsl:for-each select="components/contract/reactivity">
      <xsl:for-each select="service-response">
         <task>
            <xsl:for-each select="property">
               <xsl:attribute name="name" select="value"/>
            </xsl:for-each>
            <xsl:attribute name="name" select="fn:concat
               (name,'T')"/>
            <xsl:if test="../timeConstraint">
               <xsl:attribute name="D" select="../timeConstraint
                  /maxSafeTime"/>
            </xsl:if>
         </task>
      </xsl:for-each>
   </xsl:for-each>
</tasktable>
....
```

Figure 29: TIMES Task Transformation Rule

Issues in using XSLT for defining transformation rules

Despite all the advantages of XSLT, some limitations were encountered during the implementation of the transformation rules in XSLT, some of which are the following:

- XSLT variables cannot change their values and are only valid in context. For example, a variable assigned a value in a loop cannot be accessed either outside the loop or in the next iteration of the loop, nor can the value be modified. The only way to deal with this problem is to repeat the process of calculating the values of the variables whenever needed, as a variable can neither be stored nor reused.

- For first-time users, it is not easy to implement such complicated business rules.

4.2 TADL XML

The TADL XML file contains the specifications of component-based systems defined using TADL. The XML schema of the file conforms to the component meta-architecture discussed in Chapter 2, and represents the input to the Transformation Tool. XML was chosen for the internal representation of the system defined using our component model. What is meant by the internal textual representation is the internal format in which the model will be internally defined. This format will be hidden from the end-user and automatically generated by the Visual Modelling tool (VMT) which is used to design the component systems visually. The model's components are stored in this internal representation in the repository. The main goal of such an internal representation is to facilitate checking, design-time analysis and transformation of the system into different models, such as the behaviour model and the real-time model, and it will make it much easier to exchange information between different platforms and software products. One of the requirements of such a format is to be platform-independent. *Extensible Markup Language (XML)* is used to describe documents in a standardized text-based format [BD03]. XML is also considered to be a framework for projects involving moving information from place to place, and even software product to software product [GP02]. We decided to use XML language for the internal representation of TADL specification because: (1) it is widely supported by standards committees, tool vendors and practitioners, (2) it is platform-independent, and so not restricted to a particular hardware or network architecture, (3) linking is allowed within XML documents, which allows specific elements and attributes to be indexed, (4) off-the-shelf tools are available for constructing, editing and visualizing XML documents, and (5) there is support for modular extensibility [DvdHT05].

There are many ways to define grammars against which to validate XML documents. Two of the most popular are DTD and XML schema. DTD has less support for modular extensibility where the XML schema is much more effective, and provides an easy method for defining a construct and extending it later on [ABD+01]. Because of this, we chose to use the XML schema to define the grammar for our models.

One of the contributions of this thesis is to define the XML schemas for representing the TADL model. The main objective is to make those schemas readily understandable, extensible and reusable. This will help us to modify these XML schemas in the future if we want to add more elements and structures to our model. To achieve this, modularity was used extensively, defining related elements as complex types to increase that modularity and to improve reusability. The XML schemas constitute the basis for this work, the goal being to take a system defined using TADL model and store it in an XML format that satisfies those predefined schemas, and then perform the necessary transformations to enable execution of the design-time analysis.

TADL XML schemas

The TADL XML schemas were defined to conform to the TADL meta-model presented in Chapter 2. The definition of this schema was divided into six files to increase its maintainability and make it easier to understand. Below is a detailed discussion of the content of those files.

ComponentType file: This file contains the componentType and architecture definition. Figure 30 shows the portion of the file that contains the componentType definition. Each component may have a name, a list of properties, a list of attributes, a list of constraints, a user, a list of interfaceTypes, an architectureType, a contract and a description. Figure 31 shows the XML schema for defining architectureTypes. Each architectureType may contain a name, a list of componentTypes, a list of connectorTypes that connect components, a list of attributes, a list of constraints, a list of attachments and a description.

ConnectorType file: This file defines the connectorType that is responsible for connecting the components to one another in the architectureTypes. The content of this file can be found in Appendix A.

ContractType file: This file contains the definition of the contractType and its main elements. Each contract contains a name, a list of reactivity, a list of safety properties and a

```
...
<xs:complexType name="ComponentType">
   <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="property" type="Property" minOccurs="0"
         maxOccurs="unbounded"/>
      <xs:element name="attribute" type="Attribute" minOccurs="0"
         maxOccurs="unbounded"/>
      <xs:element name="constraint" type="xs:string"
         minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="user" type="User" minOccurs="0"/>
      <xs:element name="interfaceTypes" type="InterfaceType"
         minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="architectureType" type="ArchitectureType"
         minOccurs="0"/>
      <xs:element name="contract" type="ContractType"
         minOccurs="0"/>
      <xs:element name="discreption" type="xs:string"
         minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
...
```

Figure 30: ComponentType Schema

description. Each of those elements is defined in this file. The most important is the reactivity. Each reactivity contain a name, an ID, a service-request (stimulus), a service-response, a data constraint, a time constraint, an update element (post conditions), a select element (preconditions), a list of actions (responses) and a description. Figure 32 shows the XML schema for defining the reactivity. The full content of this file can be found in Appendix A.

InterfaceType file: This file defines the interfaceType and its main elements. Each interface may contain a name, a protocol, a list of attributes, a list of service-Types and a description. The serviceType schemas is defined in this file. Each serviceType may contain a name, an ID, type, a list of attributes, a list of constraints, a list of parameterTypes (data parameters), a list of properties and a description. Figure 33 shows the XML schema for the ServiceType. The full schema of this file can be found in Appendix A.

```
...
<xs:complexType name="ArchitectureType">
   <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="componentType" type="ComponentType"
         minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="connectorType" type="ConnectorType"
         minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="attribute" type="Attribute"minOccurs="0"
         maxOccurs="unbounded"/>
      <xs:element name="constraint" type="xs:string"minOccurs="0"
         maxOccurs="unbounded"/>
      <xs:element name="attachments" type="Attachment"
         minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="descreption" type="xs:string"
         minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
...
```

Figure 31: ArchitectureType Schema

PackageType file: This file defines the packageType that might be used to store elements for future reuse as packages. Those packages can contain interfaces, contracts, connectors or components. The full XML schema of this file can be found in Appendix A.

RBAC file: This file defines the RBAC mechanism for ensuring the security of the system. It defines the users, the privileges and the access rights of each user to the services and data parameters. The full content of this file can be found in Appendix A.

System file: This file defines the system configuration where all the previous files comes together. When a system is defined, it should conform to this XML schema. It contains the deployment rules for physical and software components and also the RBAC security definitions. The full content of this file can be found in Appendix A.

```
...
<xs:complexType name="Reactivity">
   <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="service-request" type="ServiceType"/>
      <xs:element name="service-response" type="ServiceType"/>
      <xs:element name="dataConstraint" type="DataConstrain"
         minOccurs="0"/>
      <xs:element name="timeConstraint" type="TimeConstrain"
         minOccurs="0"/>
      <xs:element name="update" type="Update" minOccurs="0"
         maxOccurs="unbounded"/>
      <xs:element name="select" type="Select" minOccurs="0"
         maxOccurs="unbounded"/>
      <xs:element name="action" minOccurs="0"
            maxOccurs="unbounded">
         <xs:complexType>
            <xs:complexContent>
               <xs:extension base="ServiceType">
                  <xs:sequence>
                     <xs:element name="from" type="xs:string"/>
                     <xs:element name="FromId" type="xs:string"/>
                     <xs:element name="to" type="xs:string"
                        minOccurs="0"/>
                  </xs:sequence>
               </xs:extension>
            </xs:complexContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="descreption" type="xs:string"
         minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
...
```

Figure 32: Reactivity Schema

4.3 Transformation Process and GUI

The component represents the implementation of the transformation process and the implementation of the GUI of the tool. Java programming language was chosen to implement this component. There were many reasons for this, some of which are: (1) the widespread popularity and simplicity of Java: one of the main concerns was that the tool be maintainable, and using a popular and simple programming language would facilitate this; and (2) Java is portable: our tool must run on different platforms without the need to recompile, and this language is platform-independent. Figure 34 shows the class diagram of this component. It contains five classes, which are the following:

- *TransformationTool class:* This is the main class of this component. It contains the implementation of the GUI and the implementation of the transformation process. It uses an external XSLT processor to perform the transformation, which is defined in the two *Java Archive* files, saxonsa.jar and xercesImpl.jar.

- *TreeBuilder class:* This class is used to define a Tree view of an XML file, which in turn is used to represent the input or output XML file in the GUI.

- *XmlFilter class:* This class is used for filtering XML files when opening or saving the input or output of the transformation tool.

- *TimesPatch class:* This class is responsible for defining a patch that is used for the transformation to the TIMES tool. The reason for having such a file is that the TIMES tool does not support the logical operator "OR". As mentioned earlier, data constraints are "predict" statements. They can contain the logical operators "AND" and "OR". Since UPPAAL supports them, there is no need to perform any special kind of transformation. By contrast, TIMES does not support them. To solve this problem, the transformation rules were modified so that, if a reactivity has a data constraint containing "OR", the reactivity is translated into multiple reactivities that have as a data constraint a statement which does not contain "OR". This modification to the transformation process is implemented in this class.

```
...
<xs:complexType name="ServiceType">
   <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="type" type="xs:string" minOccurs="0"/>
      <xs:element name="attribute" type="Attribute" minOccurs="0"
         maxOccurs="unbounded"/>
      <xs:element name="constraint" type="xs:string" minOccurs="0"
         maxOccurs="unbounded"/>
      <xs:element name="parameterType" type="ParameterType"
         minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="proprty" type="Property" minOccurs="0"
         maxOccurs="unbounded"/>
      <xs:element name="discreption" type="xs:string"
         minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
...
```

Figure 33: ServiceType Schema

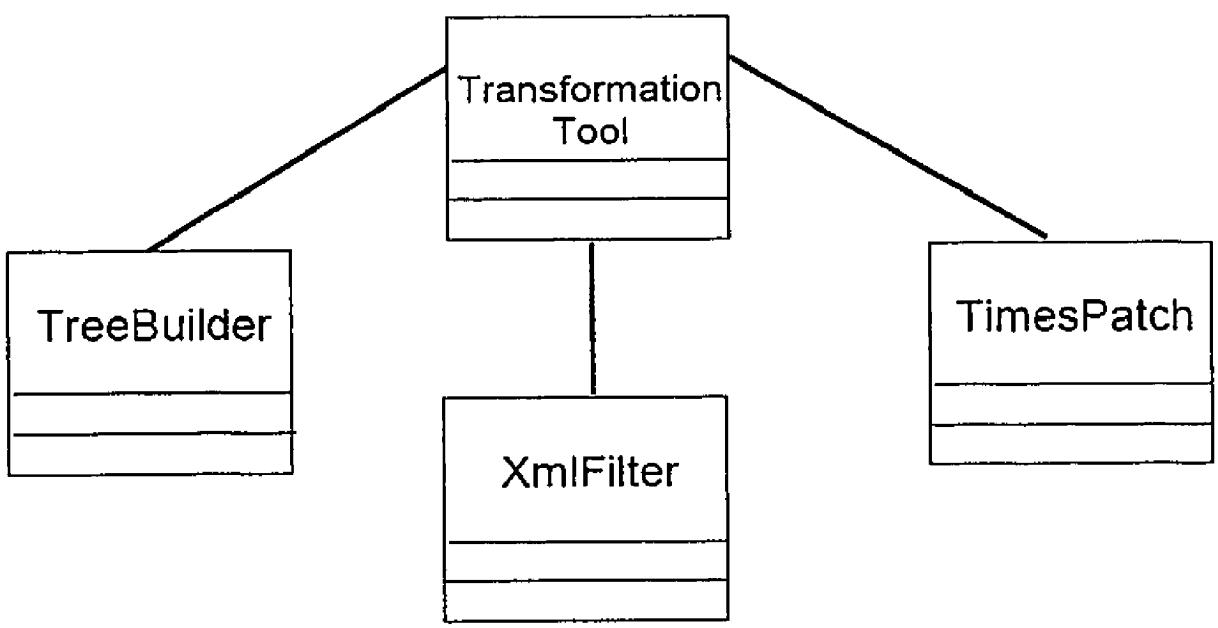

Figure 34: Transformation Procees and GUI Class Diagram

4.4 UPPAAL or TIMES XML

This component represents the output of the system. It can be either a UPPAAL XML file understood by the UPPAAL tool, or a TIMES XML file understood by the TIMES tool. Each of these files should conform to the UPPAAL or TIMES schemas. The XML DTD schema for the UPPAAL XML file is shown in Figure 35. It defines the structure of the UPPAAL XML file. The root should always be the `<nta>` XML element containing the `<declaration>`, `<template>` and `<system>` elements. The `<template>` element contains the local declaration, locations and transitions.

4.5 TransformationTool Demonstration

This section will present a simple demonstration of the TransformationTool. To run the tool, the TransformationTool.jar file is double-clicked and the tool should start. Figure 36 shows a view of the tool. It consists of the menu bar, tool bar, input panel and output panel. The tool bar contains three buttons: "Open", "Transform" and "Save". It also contains two buttons, one offering the option "To UPPAAL" and the other "To TIMES". To open an input TADL XML file, the "Open" button is clicked and a select-file window opens, as in Figure 37. The input TADL XML file can be selected and opened by clicking on it. The input will then be displayed in the input panel. There are two views for the input XML, the tree view as in Figure 39 and the text view as in Figure 38. The user can select either the "To UPPAAL" or "To TIMES" option, depending on the type of output required by the user. The default option selected is "To UPPAAL". The "Transform" button can then be clicked to perform the transformation. This button is only active when an input file is opened. After clicking the "Transform" button, the transformation is viewed in the output panel, which shows the transformation result, either in the tree view as in Figure 40 or in the text view as in Figure 41. To save the result, the user can click on the "Save" button, which opens a save-file window as in Figure 42, and select a name and the location for saving the transformation result. The saved file can be opened in UPPAAL if "To UPPAAL" was selected or in TIMES if "To TIMES" was selected.

```
<!ELEMENT nta (imports?, declaration?, template+, instantiation?,
   system)>
<!ELEMENT imports (#PCDATA)>
<!ELEMENT declaration (#PCDATA)>
<!ELEMENT template (name, parameter?, declaration?, location*,
   init?, transition*)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST name x    CDATA #IMPLIED
               y    CDATA #IMPLIED>
<!ELEMENT parameter (#PCDATA)>
<!ATTLIST parameter x    CDATA #IMPLIED
                    y    CDATA #IMPLIED>
<!ELEMENT location (name?, label*, urgent?, committed?)>
<!ATTLIST location id ID #REQUIRED
    x  CDATA #IMPLIED
     y  CDATA #IMPLIED>
<!ELEMENT init EMPTY>
<!ATTLIST init ref IDREF #IMPLIED>
<!ELEMENT urgent EMPTY>
<!ELEMENT committed EMPTY>
<!ELEMENT transition (source, target, label*, nail*)>
<!ATTLIST transition x    CDATA #IMPLIED
                     y    CDATA #IMPLIED>
<!ELEMENT source EMPTY>
<!ATTLIST source ref IDREF #REQUIRED>
<!ELEMENT target EMPTY>
<!ATTLIST target ref IDREF #REQUIRED>
<!ELEMENT label (#PCDATA)>
<!ATTLIST label kind CDATA #REQUIRED
x    CDATA #IMPLIED
y    CDATA #IMPLIED>
<!ELEMENT nail EMPTY>
<!ATTLIST nail x    CDATA #REQUIRED
               y    CDATA #REQUIRED>
<!ELEMENT instantiation (#PCDATA)>
<!ELEMENT system (#PCDATA)>
```

Figure 35: UPPAAL XML DTD

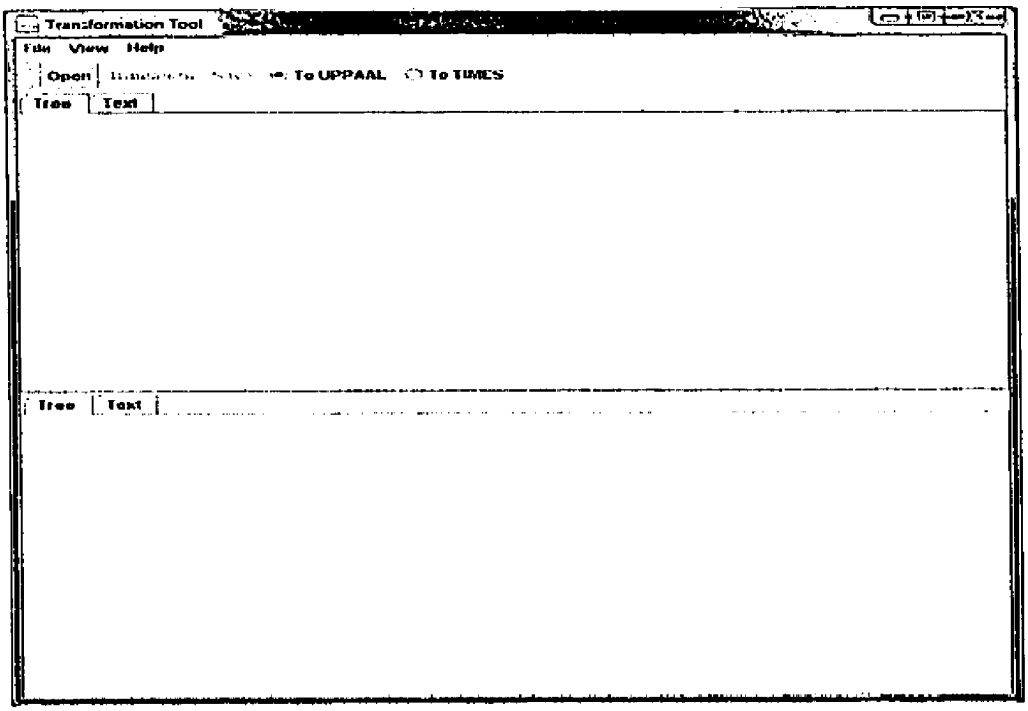

Figure 36: TransformationTool Main View

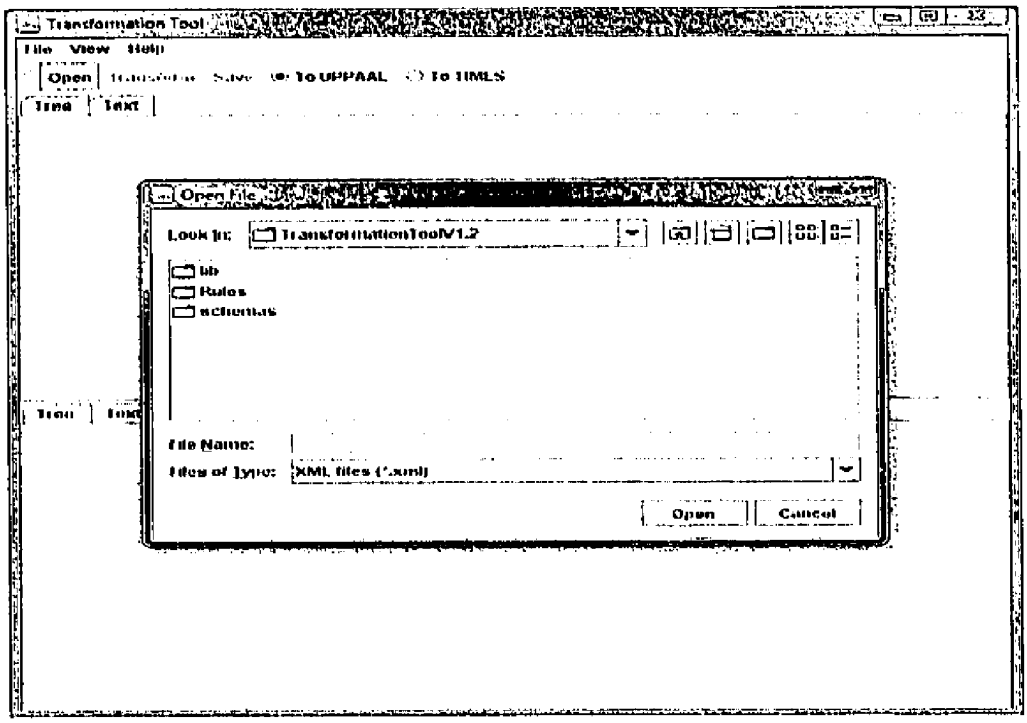

Figure 37: TransformationTool Input Open Window

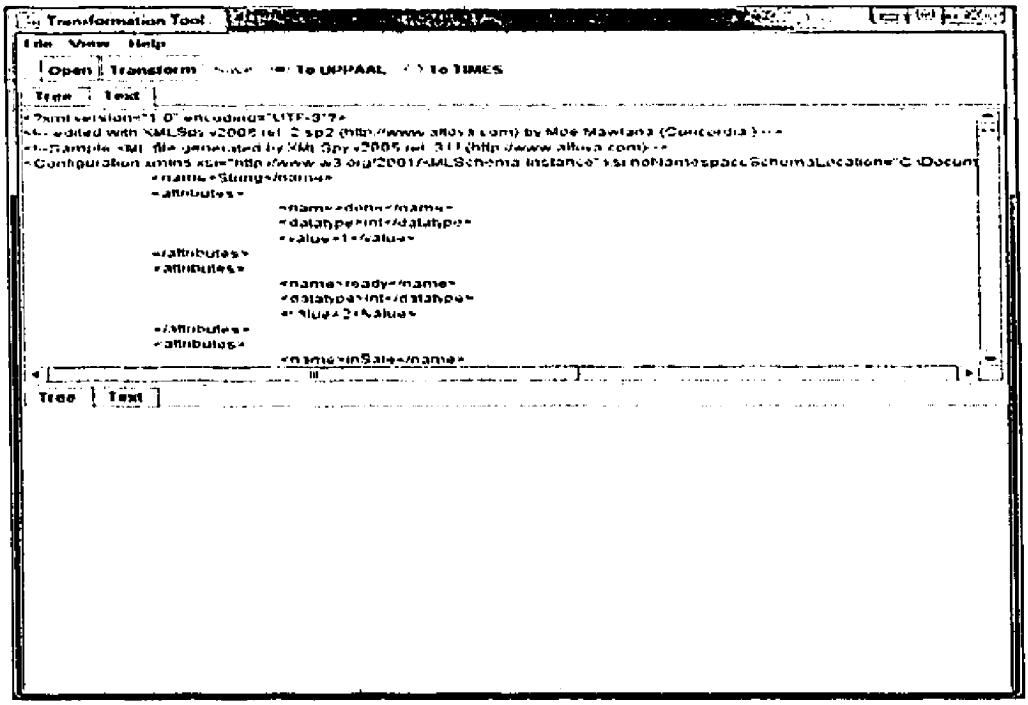

Figure 38: TransformationTool Input Text View

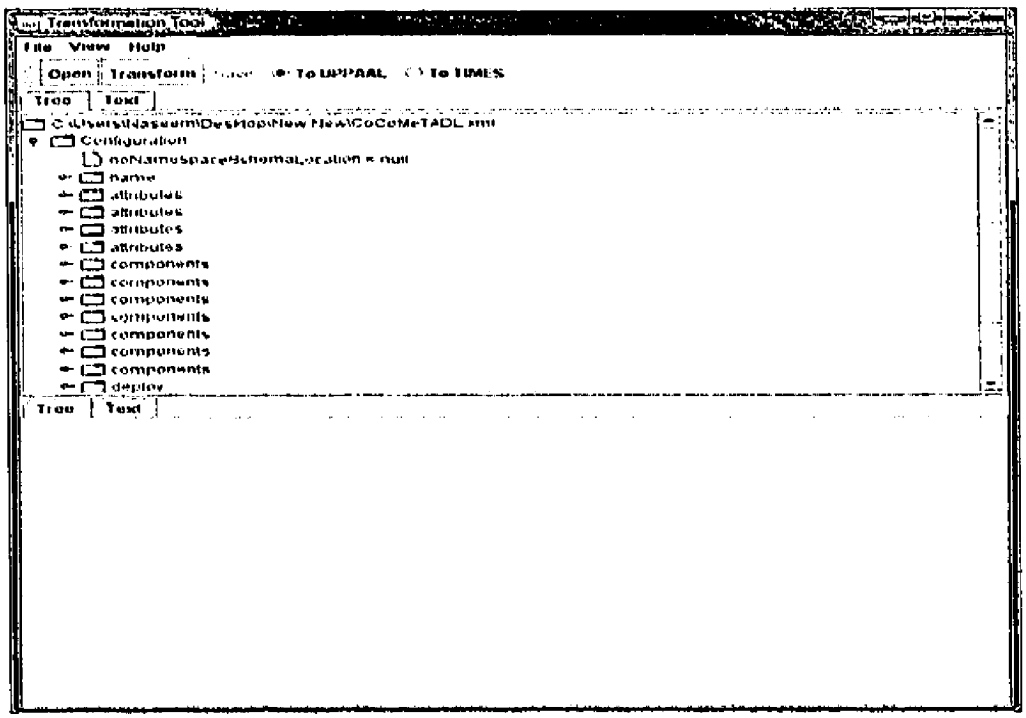

Figure 39: TransformationTool Input Tree View

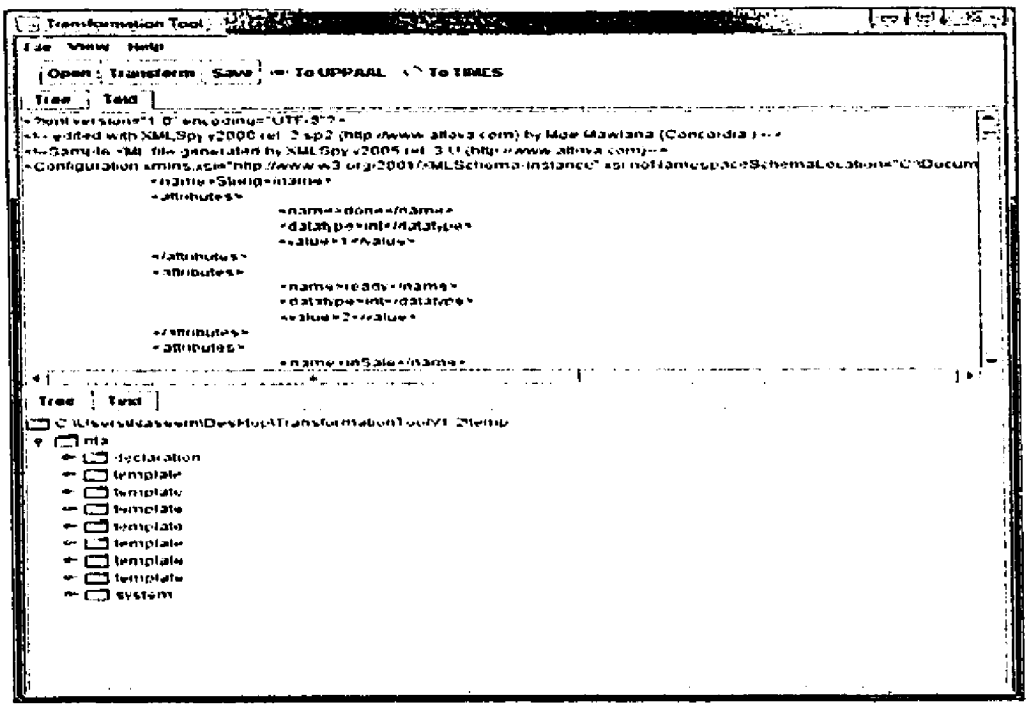

Figure 40: TransformationTool Output Tree View

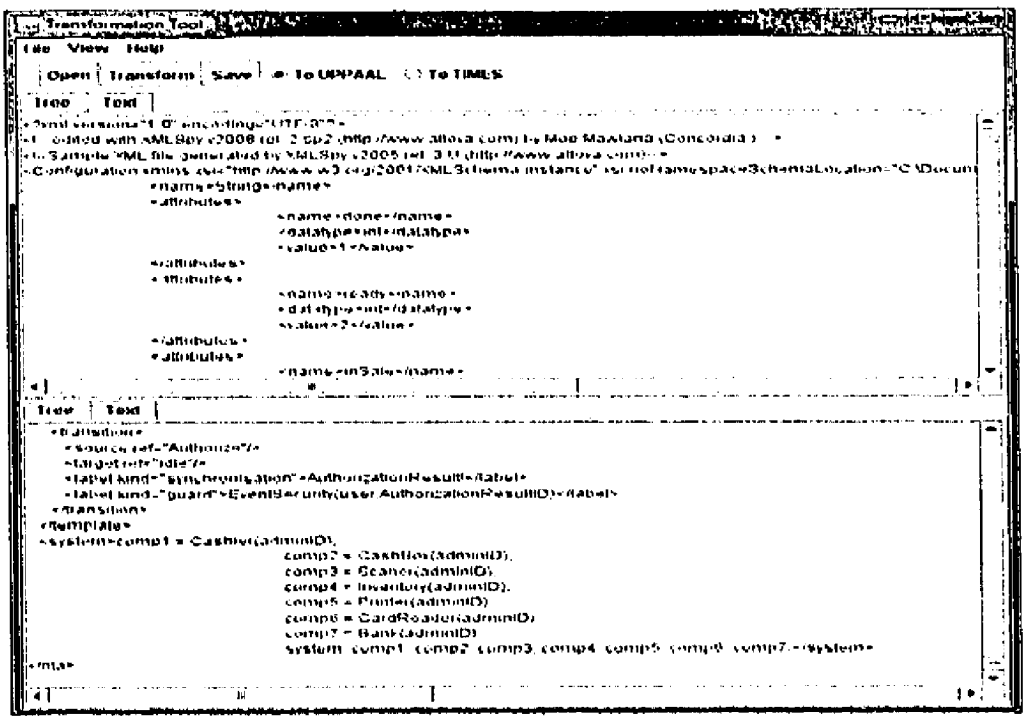

Figure 41: TransformationTool Output Text View

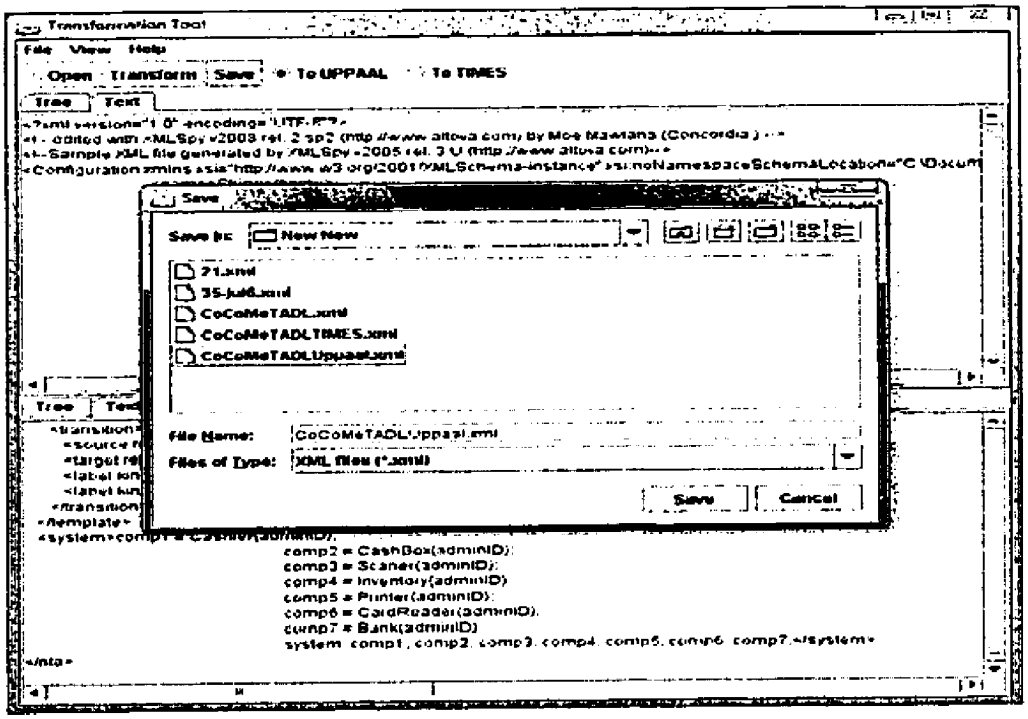

Figure 42: TransformationTool Output Save Window

4.6 Experience with UPPAAL and TIMES

In this section, I offer my personal experience with using UPPAAL and TIMES tools. While the UPPAAL tool is a mature tool that has been available for more than a decade now, it is still not perfect. TIMES tool is a much younger tool. Following are my own observations

- UPPAAL has a strange way of dealing with boolean expressions, in that it does not understand expressions with two sides. For example, in order for UPPAAL to understand the expression 1<x<4, we would need to define it as 1<x AND x<4. It is very important to acknowledge this peculiarity, because, although UPPAAL will not give you an error message if you use the first format, the logical result will be wrong.

- UPPAAL is easy to use and friendly. It has some attractive features, one of them being *generate counter example*, which is very useful when a verification test fails. In this case, UPPAAL provides an example showing how the test failed and what values caused the failure. Moreover, it helps the user to visualize it using the simulator.

- With UPPAAL, the user needs to be careful in choosing limits for the selected variables when performing the verification test, because taking too wide a range of values may cause the process to stop. For example, for one of the tests, in which I had three values, a, b and c, to which I assigned values of 100, 200 and 300 respectively, verification took more than 10 hours and I did not obtain a result. I repeated the test more than once, and still did not obtain a result. I then replaced the values with a=10, b=20 and c=30, and a result was returned in a couple of minutes. It is very important to bear this limitation in mind when working with UPPAAL.

- UPPAAL does not accept spaces in template names, so it is important not to use them in component names.

- UPPAAL will not return an error message if two variables with the same name are defined in the global and local declarations. The tool will sometimes use the value defined in the local declaration and sometimes the value defined in the global declaration, so it is imortant not to use a name for a variable more than once.

- UPPAAL does not allow use of a name for an element more than once, even if it applies to different elements. For example, the tool will reject a channel name and a location name if they are the same. We need to make sure to name elements in a unique way, such as adding an "S" after each location (state) name.

- TIMES does not define the logical operation OR. This is a problem, for example, when data constraints contain predict statements containing OR.

- Syntax error messages returned by TIMES are very useful.

- TIMES is more complicated to use than UPPAAL; moreover, it is easier to define states and edges in UPPAAL than in TIMES.

4.7 Summary

In this chapter, the implementation of the TransformationTool has been presented, and the implementation of all four components of the tool design have been described. In addition, a demonstration of the tool was provided. To conclude, a brief summary of the author's personal experience with the TIMES and UPPAAL tools was presented.

Chapter 5

Case Study

This Chapter presents two case studies that are used to illustrate the transformation process from a system defined using TADL model to UPPAAL and TIMES models. First, a description of each case study is presented. Then, the TADL presentation of this case study is presented. The UPPAAL representation of the first case study and the TIMES representation of the second case study are also presented. Finally, the verification performed using UPPAAL and TIMES is presented. Due to space limitation the TADL XML of the case studies will not be presented here, it can be found in [Ibr08].

5.1 Common Component Modelling Example - CoCoME

A common component modelling example (CoCoME) has been introduced by the component development community [HKW+08] to be used by different component models to evaluate and compare the practical application of existing component models using a common component-based system as a modelling example. One of the contributions of the work presented in this thesis is the application of the component-based model for developing a trustworthy RTRS on this common example. Below, we introduce this case study and show how it was applied using the TADL model.

5.1.1 Introduction

The CoCoME defines the Trading System, which is concerned with all aspects of handling sales at a supermarket, including the interaction with the customer at the cash desk (product scanning and payment) and recording of the sale for inventory purposes. It also deals with ordering goods from wholesalers and generating various kinds of reports.

5.1.2 System overview

We begin the system overview with the cash desk. Figure 43, which was extracted from [HKW+08], shows an overview of the parts of the cash desk, where, the customer pays for the products he wants to buy. An express checkout is available to customers with only a few items, to speed up the transaction process. The cash desk consists of the following:

- a *Cash Box*, which begins and ends the transaction, and holds cash received from customers;
- a *Bar Code Scanner*, which is used to identify the products being purchased;
- a *Card Reader*, which handles card payments (cash payments are handled by the Cash Box);
- a *Printer*, for printing the bill to be handed to the customer at the end of the transaction;
- a *Light Display*, to signal to the customer whether the cash desk is currently operating in normal or in express mode;
- a *Cash Desk PC*, for handling the transaction and communicating with the *Bank*.

Each store has multiple cash desks. Also, each store has its own *Store Server* and a *Store Client* which are connected. The Store Client is used by the manager to view reports, order products, and administer the inventory. Each store is connected to an *Enterprise Server*.

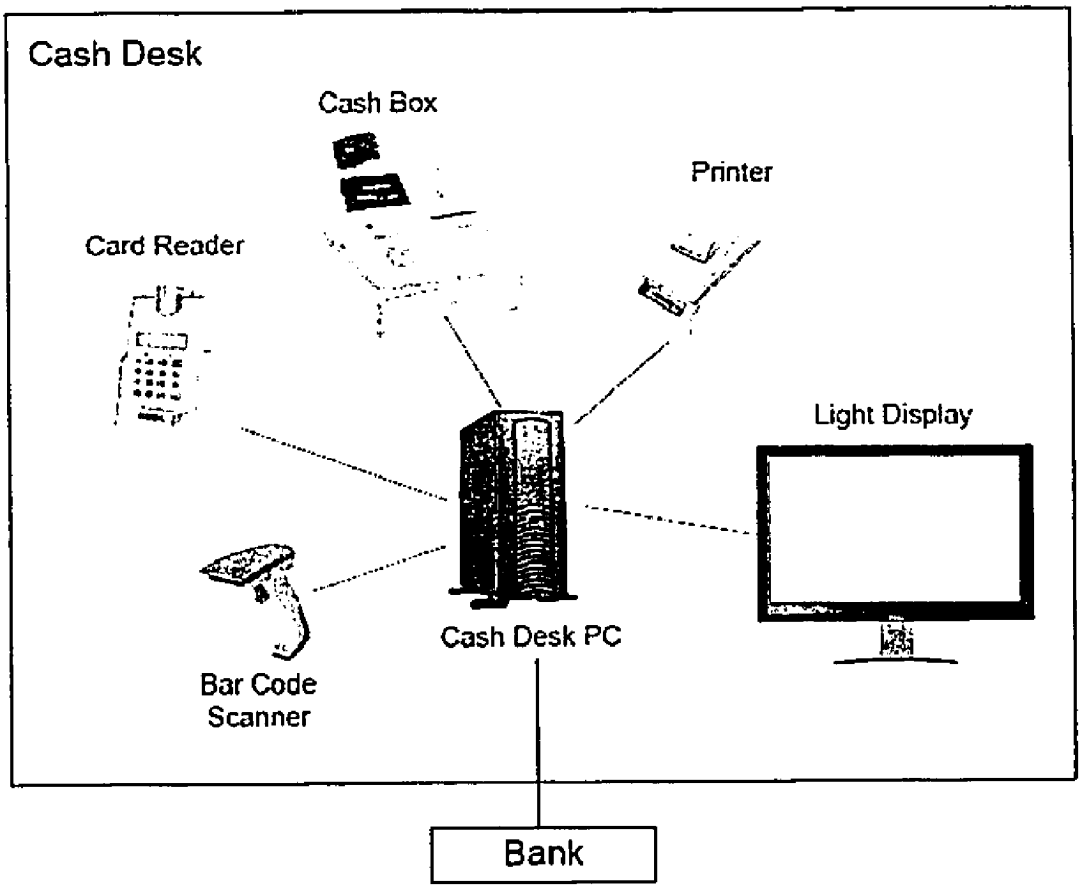

Figure 43: Cash Desk

5.1.3 System Requirements

This section introduces the functional and non-functional requirements of the Trading System. A more detailed presentation of these requirements and the related Use Cases can be found in [HKW+08].

Process Sale

The system has to handle the purchase transaction at a cash desk. The customer arrives at the cash desk with items to purchase. The cashier begins the new sale process by pressing the *Start New Sale* button at the Cash Box. The cashier then enters the item identifier using the Bar Code Scanner. The system takes the item identifier and returns the product description, price and running total. These steps are repeated until all the items have been

scanned, at which point the cashier presses the *Sale Finished* button and proceeds to the payment options. The cashier can either press the *Cash Payment* button to initiate a cash payment, or the *Card Payment* button to initiate a card payment. In the case of a cash payment, the customer hands the cash to the cashier and the cashier returns the change, if any, and ends the sale. In the case of a card payment, the customer hands the card to the cashier, who pulls it through the Card Reader and waits for the Bank to validate the payment. Finally, the system logs the completed sale information in the inventory, and the receipt is printed and handed to the customer. The time requirements for processing the sale are the following:

- Time for pressing the *Start New Sale* button, 1.0 s;
- Time for scanning an item, 5.0 s;
- Time for showing product description, price and running total, 1.0 s;
- Time for pressing the *Sale Finished* button, 1.0 s;
- Time for processing a cash payment, 120.0 s;
- Time for pressing the *Card Payment* button, 1.0 s;
- Time waiting for validation, 30.0 s;
- Time for updating the inventory, 2.0 s;
- Time for printing the receipt and handing it to the customer, 3.0 s.

Manage Express Checkout

The system should be able to change Cash Desks to express mode. This is done by checking that the condition for express mode is fulfilled, which is that 50% of the sales in the preceding 60 minutes be for fewer than 8 items. The change to express mode should proceed automatically and the cashier should have the option of disabling it. The time requirements for managing an express checkout are the following:

- Time for switching the light display, 1.0 s;
- Time for pressing the *Disable Express Mode* button, 1.0 s;
- Time for switching to express mode, 1.0 s.

Order Products

The system should permit managers to order products by producing a list of all the products available and a list of the items running out of stock. The manager selects the products he wants to order, enters the corresponding quantities and then presses the *Order* button on the Store Client. The system chooses the suppliers and sends them the orders, at which point the order details are displayed. The time requirements for ordering products are as follows:

- Time to wait before the lists of all products and missing products are shown, 1.0 s;
- Time for choosing the products to order and entering the quantities, 10.0 s.

Receive Products Ordered

The system should remember what new products have been ordered. When the products arrive, the stock manager checks the identifiers attached to those orders against the list of products ordered to ensure that the delivery is complete and correct. If it is, he enters the order identifiers and presses the *Roll in received order* button. The system then updates the inventory. The time requirements for receiving ordered products are as follows:

- Time for pressing the *Roll in received order* button, 1.0 s;
- Time for updating the inventory, 1.0 s;

Show Stock Reports

The system should be able to provide stock-related reports. These are created by the manager entering the store identifier and pressing the *Create Report* button. The system then provides a report on all the available stock items in the store. The time requirements for showing stock reports are as follows:

- Time for entering store ID and pressing the *Create Report* button, 1.0 s;
- Time for generating the report, 1.0 s

Show Delivery Reports

The system should calculate the mean time for delivery to an enterprise. A delivery report is created by the manager by entering the supplier ID and pressing the *Create Report* button. The system generates a report showing the mean time to delivery for a specific supplier. The time requirements for producing delivery reports are as follows:

- Time for entering supplier ID and pressing the *Create Report* button, 1.0 s;
- Time for generating the report, 1.0 s.

Change Price

The system should enable the manager to change the price of an item. The system presents a list of all the available items. The manager selects a product and changes its price and confirms that price by pressing ENTER. The time requirements for changing the price of a product are as follows:

- Time for generating the overview, 1.0 s;
- Time for selecting a product and pressing ENTER, 6.0 s;
- Time for changing the price in the inventory, 5.0 s.

Product Exchange

If a store runs out of a product, the system should provide the functionality required for the store to send a request to the Enterprise Server with the product ID. The Enterprise Server will ask all stores that are less than 300 km away to update their data in the Enterprise Server, so that the enterprise can make the necessary calculations to determine which store has the available product and whether or not it is economically feasible to send the product

from one store to another. The time requirements for the product exchange functionality are as follows:

- Time for the Store Server to query the Enterprise Server, 2.0 s;
- Time for the Enterprise Server to query one Store Server, 2.0 s;
- Time for flushing the cache of one Store Server and returning the result, 2.0 s;
- Time for determining from which store to deliver, 1.0 s;
- Time for marking goods as incoming on the Store Server, 2.0 s;
- Time for sending a delivery request to the Store Server, 2.0 s.

Security Requirements

The system should satisfy the following security requirements:

- Only the Manager can request changes to product prices. Any attempt by the cashier to do so is denied.
- Only the Manager can order new products. Any attempt by the cashier to do so is denied.
- Only the Manager can request delivery reports. Any attempt by the cashier to do so is denied.
- Only the Manager can request stock reports. Any attempt by the cashier to do so is denied.

5.1.4 TADL representation

The TADL representation begins by presenting the component diagram. From the previous description of the CoCoME case study, it can be concluded that the system has a sub-system which represents the store system. Figure 44 presents the components of the store system.

Those components are: Cash Box, Printer, Bar Code Scanner, Card Reader, Bank, Light Display, Cashier, Store Client, Manager, Stock Manager and Inventory. The Cash Box, Printer, Bar Code Scanner, Card Reader, Light Display and Cashier components represents the Cash Desk. The store systems are connected to the enterprise inventory, and together they create the Trading System, as depicted in Figure 45. Before going into the details of each component, it is important to understand that global-level variables are needed to model the system. These variables are introduced as system-level attributes, and are listed in Table 1.

Below is a detailed description of the TADL representation of the above components. The focus will be on the contract part of each component and their reactivities, as they are the only parts included in the transformation process because they define the behaviour of the component.

Cash Box

This component performs the selling operation initiated by the cashier. It provides the following services:

- *Input Services :* PassItem, BarCode, Cash, Card, Approved, Info, SaleFinished, Declined, CheckIfExpress, YesExpress, NotExpress and DisableExpress.
- *Output Services :* Scan, GetInfo, ReadCard, Print, Pay, CheckLastHour, TurnLightOn, AddToInventory, IsMoreItem and TurnLightOff.
- *Internal Services :* ReturnChange, AddTotal, Ignore and ChangeModeToNormal.

These services have data parameters which are used to hold information sent and received between components. The data parameters for the Cash Box services can be seen in Table 2. The Cash Box component contract has 12 reactivities. Table 3 shows the main elements of each reactivity. The request represents the service requested (stimulus). The response represents the corresponding service response. The data constraint (DC) represents the precondition for this reactivity. The time constraint (TC) presents the time conditions of the

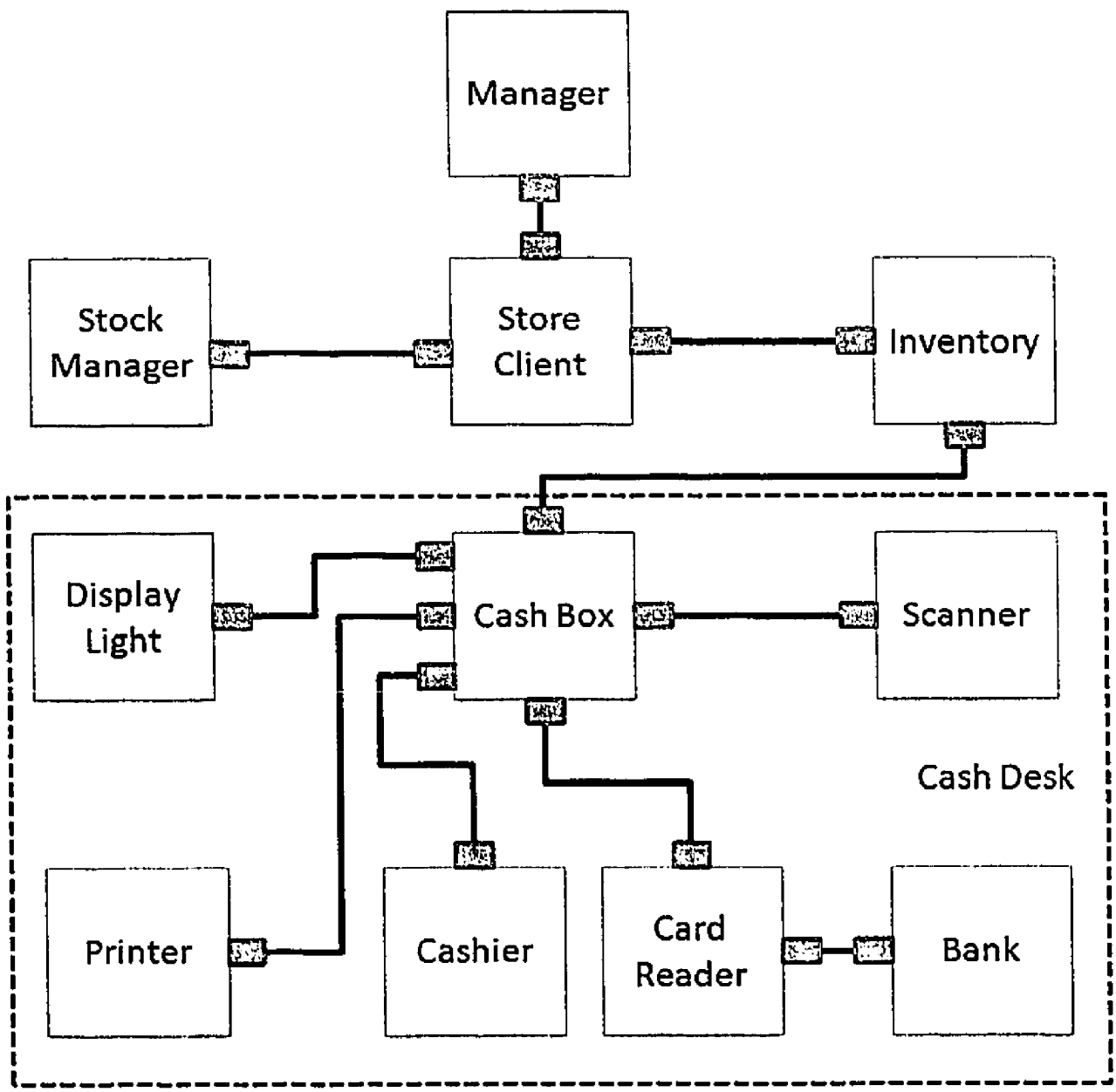

Figure 44: Store System Components

Variable	Type	Description
Mode	int	Used to control the mode of the Cash Box.
Ready	int	Used to represent the ready mode of the Cash Box.
inSale	int	Used to represent the mode of the Cash Box when it is in a sale process.
Waiting	int	Used to represent the waiting mode of the Cash Box.
disable	int	Used to represent the disable mode of the Cash Box when it is in the process of disabling the express mode.
isExpress	int	Used to represent the operation of the Cash Box, which can either be in express or in normal mode.
Express	int	Used to represent the express operation of the Cash Box.
normal	int	Used to represent the normal operation of the Cash Box.
NewMode	int	Used to control the mode of the Store Client and it is defined as an integer variable.
Readyy	int	Used to represent the ready mode of the Store Client.
itemOrder	int	Used to represent the mode of the Store Client when it is in the middle of an item ordering process.
report	int	Used to represent the mode of the Store Client when it is in the middle of an report ordering process.
deliveryReport	int	Used to represent the mode of the Store Client when it is in the middle of a delivery report ordering process.
priceChange	int	Used to represent the mode of the Store Client when it is in the middle of the price-changing process.
rollIn	int	Used to represent the mode of the Store Client when it is in the middle of rolling in orders that have been arraived in the inventory.

Table 1: System-level Variables

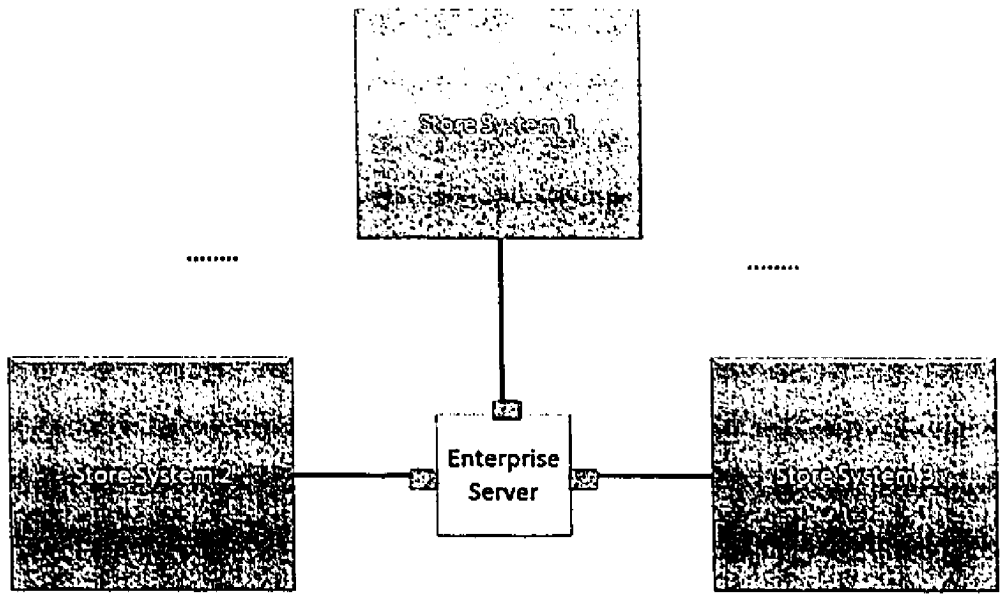

Figure 45: Trading System

Service	Data Parameter	Notes
BarCode	Code:int	Holds the bar code number of the scanned item
GetInfo	Code:int	Holds the bar code number of the scanned item
Approved	transactionNum:int	Holds the authorization results
Declined	transactionNum:int	Holds the authorization results
Print	SaleInfo:string	Holds the sale information
AddToInventory	SaleInfo:string	Holds the sale information
Info	ItemInfo:string	Holds the information about the item, for example its name and price
CheckLastHour	CashBoxNumber:int	Holds the number of the cash box

Table 2: Data Parameters for the Services of the Cash Box Component

Request	Response	DC	TC	Update	Actions
PassItem	Scan				
BarCode	GetInfo			Mode:=done	AddToInventory, Print
Cash	ReturnChange		120s		
Card	ReadCard		1s		
Approved	Print			Mode:=done	AddToInventory
Info	AddTotal		1s		IsMoreItem
SaleFinished	Pay		1s		
Declined	Pay				
CheckIfExpress	CheckLastHour	Mode==done		Mode:=waiting	
YesExpress	TurnLightOn		1s	Mode:=ready, isExpress:=express	
NotExpress	Ignore			Mode :=ready, isExpress :=normal	
DisableExpress	TurnModeToNormal		1s	isExpress := normal	TurnLightOff

Table 3: Cash Box Component Reactivities

reactivity. The update presents the post-conditions and the actions represent other services triggered by this reactivity.

Cashier

This component represents the cashier who operates the Cash Box and is responsible for scanning the items to be sold, completing the payments, either by cash or by card, and managing the Cash Box mode (express or normal). This component needs two local-level variables, which will be defined as component-level attributes, to control the operation. Those variables are: isMore which is used to decide whether or not there are more items to be scanned; and paymentMethod which is used to decide on the method of payment (cash or card). The services defined in this component are as follows:

- *Input Services:* IsMoreItem and Pay.
- *Output Services:* PassItem, SaleFinished, Cash, Card and DisableExpress.

Request	Response	Data Constraint	TC	Update
StartSale	PassItem	Mode==ready	1s	Mode:=inSale
IsMoreItem	SaleFinished	isMore==1		
IsMoreITem	PassItem	isMore==2		
Pay	Cash	paymentMethod==1 OR isExpress == express		
Pay	Card	paymentMethod==2 OR isExpress == normal		
CancelExpress	DisableExpress	Mode ==ready		Mode :=disable

Table 4: Cashier Component Reactivities

- *Internal Services:* StartSale and CancelExpress.

The reactivities of this component can be seen in Table 4.

Scanner

This component is responsible for scanning the items to be sold and reading their bar code. It offers two services: Scan and BarCode. Scan is an input service (stimulus) and BarCode is an output service. The BarCode service has one data parameter, which is *Code* of the `int` *type*, and it holds the bar code number of the item scanned. Table 5 shows the only reactivity it has.

Request	Response	Time Constraint
Scan	BarCode	5s

Table 5: Scanner Component Reactivity

Printer

This component is responsible for printing the sale receipt. It offers two services, which are: Print and Printed. Print is an input service (stimulus) and Printed is an internal service representing the printing operation. The Print service has one data parameter, which is

Request	Response	Time Constraint
Print	Printed	3s

Table 6: Printer Component Reactivity

Service	Data Parameter	Notes
AuthorizationResult	Result:string	Holds the result of authorization process
Approved	transactionNum:int	Holds the authorization results
Declined	transactionNum:int	Holds the authorization results
Authorization	CardInfo:string	Holds the information of the card being used for the payment

Table 7: Data Parameters for the Services of the Card Reader Component

SaleInfo of the `string` *type*, and it holds the sale information. Table 6 shows the only reactivity it has.

CardReader

This component is responsible for managing the card payment process. It will read the card information and send it to the Bank for payment approval. This component offers the following services:

- *Input Services:* ReadCard and AuthorizationResult.
- *Output Services:* Authorize, Approved, and Declined.

These services have data parameters that are used to hold information sent and received between components. The data parameters for the Card Reader services can be seen in Table 7. The Card Reader component defines one local variable, called authorization, that is used to hold the result of the authorization process. Table 8 shows the reactivities that this component has.

Request	Response	Data Constraint	Time Constraint
ReadCard	Authorize		2 s
AuthorizationResult	Approved	authorization==1	
AuthorizationResult	Declined	authorization==2	

Table 8: CardReader Component Reactivities

Request	Response	Time Constraint
Authorize	AuthorizationResult	30s

Table 9: Bank Component Reactivities

Request	Response	Time Constraint
RecieveOrder	RollIn	1s

Table 10: StockManager Component Reactivity

Bank

This component represents the financial institution that is responsible for performing and approving the card payment. It offers two services, which are: Authorize and AuthorizationResult. Authorize is an input service (stimulus) and AuthorizationResult is an output service representing the result of the authorization process. The Authorize service has one data parameter, *CardInfo*, of the `string` *type*, which holds the card information of the used for payment. The service AuthorizationResult has a data parameter, *result*, of the `string` *type*, which is used to hold the authorization result. Table 9 shows the only reactivity it has.

StockManager

This component represents the stock manager, which is responsible for receiving arriving orders, checking them and rolling them into the inventory. It offers two services, which are: ReceiveOrder and RollIn. ReceiveOrder is an internal service and RollIn is an output service. The RollIn service has one data parameter, *orderNum* of the `int` *type*, which holds the number of the order being rolled into the inventory. Table 10 shows the only reactivity it has.

DisplayLight

This component represents the display light above the Cash Box which is used to indicate whether or not this Cash Box is in express mode. This component offers the following services:

Request	Response	Time Constraint	DC	Update
TurnLightOn	On	1s		
TurnLightOff	Off	1s	Mode == disable	Mode := ready

Table 11: DisplayLight Component Reactivities

- *Input Services:* TurnLightOn and TurnLightOff.
- *Internal Services:* On and Off.

Table 11 shows the reactivities that this component has.

EnterpriseServer

This component represents the enterprise server. Each Trading System has only one enterprise server, which is responsible for managing a stock item being transferred between a store which is running out of the item and another store in which the item is available. This component offers the following services:

- *Input Services:* RequestItems and NewInventory.
- *Output Services:* Incoming, UpdateInventory and SendToStore.
- *Internal Service:* FindNearStore, Process and updateInternalData.

The data parameters for these services can be seen in Table 12. Table 13 shows the reactivities that this component has.

Manager

This component represents the store manager, who can request reports and change the price of items in the inventory. This component offers the following services:

- *Input Services:* ShowList and ShowItemsPrice.
- *Output Services:* ShowItem, SelectAndOrder, CreateReport, GetDeliveryReport, ItemsPrice and SelectAndChange.

Service	Data Parameter	Notes
RequestItems	ItemNum:int	Holds the ID of the item being requested from the Enterprise Server
RequestItems	StoreNum:int	Holds the ID of the store requesting the item
FindNearStore	StoreNum:int	Holds the ID of the store that needs to find the item within 300 km
Incoming	ItemNum:int	Holds the ID of the item that will be coming in order to change its status to incoming instead of low
NewInventory	InventoryInfo:string	Holds the updated version of the store inventory
NewInventory	StoreNum:int	Holds the ID of the store that is updating its inventory on the Enterprise Server
SendToStore	ItemNum:int	Holds the ID of the item being sent to the store
SendToStore	Quantity:int	Holds the quantity being requested
SendToStore	StoreNum:int	Holds the ID of the store requesting the item

Table 12: Data Parameters for the Services of the Enterprise Server Component

Request	Response	Data Constraint	TC	Update	Action
RequestItems	FindNearStore	EnterpriseMode ==ready	2s	EnterpriseMode :=waiting	UpdateInventory
NewInventory	updateInternalData		2s	EnterpriseMode :=visable	
Process	Incoming	EnterpriseMode ==visable	2s	EnterpriseMode :=ready	SendToStore

Table 13: EnterpriseServer Component Reactivities

Service	Data Parameter	Notes
ShowList	ItemNum:int[]	Holds the IDs of the items available in the store
ShowList	ItemQuantity:int[]	Holds the quantities of the items available in the store
SelectAndOrder	ItemNum:int[]	Holds the IDs of the items selected by the manager
SelectAndOrder	ItemQuantity:int[]	Holds the quantities of the items ordered by the manager
ShowItemsPrice	ItemNum:int[]	Holds the IDs of the items available in the store
ShowItemsPrice	Price:int[]	Holds the prices of the items available in the store
AddToInventory	SaleInfo:string	Holds the sale information
SelectAndChange	ItemNum:int[]	Holds the IDs of the items changed by the manager
SelectAndChange	NewPrice:int[]	Holds the prices of the items changed by the manager
GetDeliveryReports	SupplierNum:int	Holds the ID of the supplier to whom the delivery reports belong

Table 14: Data Parameters for the Services of the Manager Component

- *Internal Service:* Order, Reports, DReports and ChangePrices.

These services have data parameters that are used to hold information sent and received between components. The data parameters for the Cash Box services can be seen in Table 14. Table 15 shows the reactivities that this component has.

StoreClient

This component represents the store client, which is used by the store manager to perform the required operations, such as changing item prices or ordering items. The store client services are:

- *Input Services:* ListOfItem, DR, GetDeliveryReport, ItemsPrice, Price, Report SelectAndChange, CreateReport, SelectAndOrder and ShowItem.

- *Output Services:* ShowList, GetItemsPrice, ShowItemsPrice, ChangePrice, GetReport and GetItemList.

Request	Response	Data Constraint	TC	Update
Order	ShowItem	NewMode ==readyy		NewMode :=itemOrder
ShowList	SelectAndOrder	NewMode ==itemOrder	1s	
Reports	CreateReports	NewMode ==readyy	1s	NewMode :=report
DReports	GetDeliveryReports	NewMode ==readyy	1s	NewMode :=deliveryReport
ChangePrices	ItemsPrice	NewMode ==readyy		NewMode :=priceChange
ShowItemsPrice	SelectAndChange	NewMode ==priceChange		

Table 15: Manager Component Reactivities

- *Internal Service:* Show, GetDR, ShowTheReports and OrderSeleceted AndShowID.

These services have data parameters which can be seen in Table 16. Table 17 shows the reactivities that this component has.

Inventory

This component represents the store server (Inventory), which stores all the information for that specific store, including product information and stock levels, delivery reports, transaction records, and so on. The Inventory component offers the following services:

- *Input Services:* GetInfo, AddToInventory, ChangePrice, GetItemsPrice, RollInInv, GetItemList, GetReport, GetDR, CheckLastHour, UpdateInventory, Incoming and SendToStore.

- *Output Services:* Info, Price, ListOfItem, Report, DR, Yes, No, RequestItems and NewInventory.

- *Internal Service:* InfoAdded, PriceChanged, RollItI, RequestItemFromEnterprise, setIncoming and sendIt.

Tables 18 and 19 shows the reactivities and data parameters the Inventory component has.

Service	Data Parameter	Notes
RollIn	orderNum:int	Holds the ID of the order being added to the inventory
RollInInv	orderNum:int	Holds the ID of the order being added to the inventory
ShowList	ItemNum:int[]	Holds the IDs of the items available in the store
ShowList	ItemQuantity:int[]	Holds the quantities of the items available in the store
GetDeliveryReports	SupplierNum:int	Holds the ID of the supplier to whom the delivery reports belong
GetDR	SupplierNum:int	Holds the ID of the supplier to whom the delivery reports belong
DR	DeliveryReports:string	Holds the delivery reports
SelectAndOrder	ItemNum:int[]	Holds the IDs of the items selected by the manager
SelectAndOrder	ItemQuantity:int[]	Holds the quantities of the items ordered by the manager
ShowItemsPrice	ItemNum:int[]	Holds the IDs of the items available in the store
ShowItemsPrice	Price:int[]	Holds the prices of the items available in the store
SelectAndChange	ItemNum:int[]	Holds the IDs of the items changed by the manager
SelectAndChange	NewPrice:int[]	Holds the prices of the items changed by the manager
Reports	SalesReports:string	Holds the sale reports
Price	ItemNum:int[]	Holds the list of IDs of all items in the inventory
Price	Price:int[]	Holds the list of prices of all items in the inventory
ChangePrice	ItemNum:int[]	Holds the IDs of the items changed by the manager
ChangePrice	NewPrice:int[]	Holds the prices of the items changed by the manager

Table 16: Data Parameters for the Services of the Store Client Component

Request	Response	Update	Time Constraint
ListOfItem	ShowList		
DR	Show	NewMode :=readdy	
GetDeliveryReport	GetDR		
ItemsPrice	GetItemsPrice	NewMode :=readyy	
Price	ShowItemsPrice		
SelectAndChange	ChangePrice		6s
Reports	ShowTheReports	NewMode :=readyy	
CreateReport	GetReport		
SelectAndOrder	OrderSelectedAndShowID	NewMode :=readyy	10s
ShowItem	GetItemList		
RollIn	RollInInv		

Table 17: StoreClient Component Reactivities

Request	Response	Update	Time Constraint
GetInfo	Info		
AddToInventory	InfoAdded		2s
ChangePrice	PriceChanged	NewMod :=readyy	5s
GetItemsPrice	Price		1s
RollInInv	RollItIn	NewMode :=readdy	1s
GetItemList	ListOfItem		
GetReport	Report		1s
GetDR	DR		1s
CheckLastHour	YesExpress		
CheckLastHour	NotExpress		
RequestItemsFromEnterprise	RequestItems		
UpdateInventory	NewInventory		2s
Incoming	setIncoming		1s
SendToStore	SendIt		2s

Table 18: Inventory Component Reactivities

Service	Data Parameter	Notes
GetInfo	Code:int	Holds the bar code number of the item
Info	ItemInfo:string	Holds item information like name and price
ChangePrice	ItemNum:int[]	Holds the IDs of the items changed by the manager
ChangePrice	NewPrice:int[]	Holds the prices of the items changed by the manager
Price	ItemNum:int[]	Holds the list of IDs of all items in the inventory
Price	Price:int[]	Holds the list of prices of all items in the inventory
RollInInv	orderNum:int	Holds the ID of the order being rolled into the inventory
GetDR	SupplierNum:int	Holds the ID of the supplier to whom the delivery reports belong
DR	DeliveryReports:string	Holds the delivery reports
RequestItems	ItemNum:int	Holds the ID of the item being requested from the Enterprise Server
RequestItems	StoreNum:int	Holds the ID of the store requesting the item
Incoming	ItemNum:int	Holds the ID of the item that will be coming in order to change its status to incoming instead of low
NewInventory	InventoryInfo:string	Holds the updated version of the store inventory
NewInventory	StoreNum:int	Holds the ID of the store that is updating its inventory on the Enterprise Server
SendToStore	ItemNum:int	Holds the ID of the item being sent to the store
SendToStore	Quantity:int	Holds the quantity being requested
SendToStore	StoreNum:int	Holds the ID of the store requesting the item

Table 19: Data Parameters for the Services of the Inventory Component

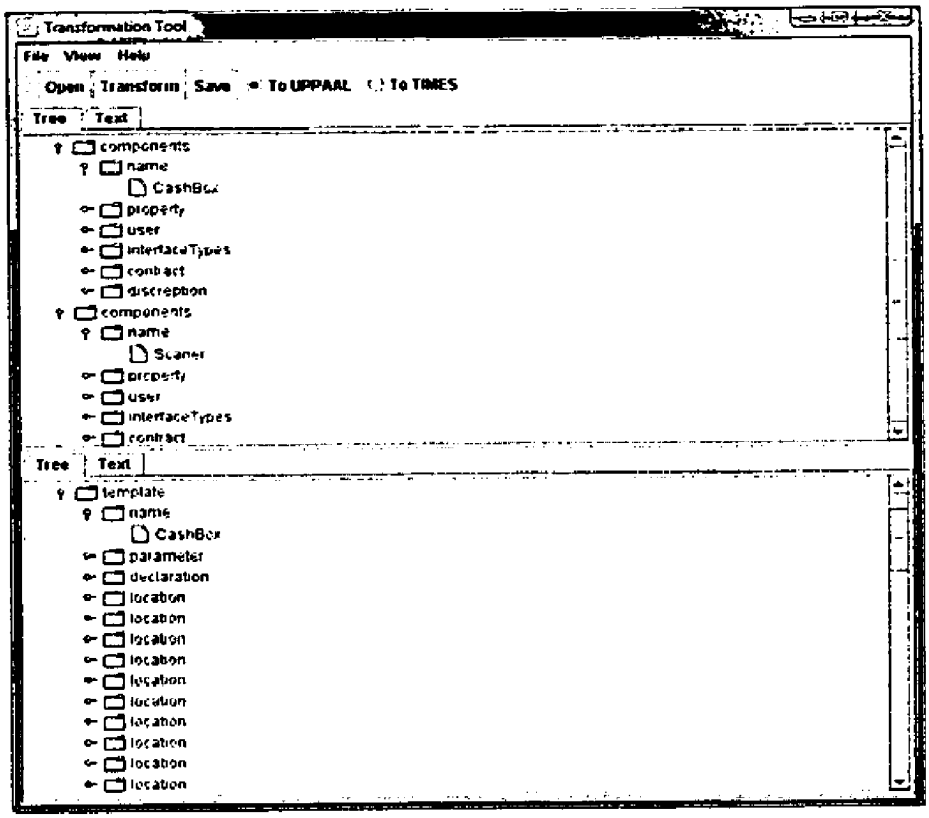

Figure 46: Transformation of CoCoME TADL to UPPAAL

5.1.5 UPPAAL Representation

The TADL XML representation of the CoCoME case study was defined in a single XML file. This file was passed to the TransformationTool, which produced the UPPAAL representation of the system. Figure 46 shows a snapshot of the TransformationTool during the transformation process. The resulting XML file was opened in the UPPAAL tool. Below is a brief preview of the resulting UPPAAL system.

Global Declaration: Figure 47 shows the global declaration of the CoCoME system that was automatically generated by the transformation process. It contains the declaration of the channels, the global-level variables, Data Security function and Even Security function.

Cash Box template: The Cash Box component was discussed earlier in the TADL representation. As a result of the transformation, the Cash Box template was automatically created and represents the Cash Box component. Figure 49 shows a view of the

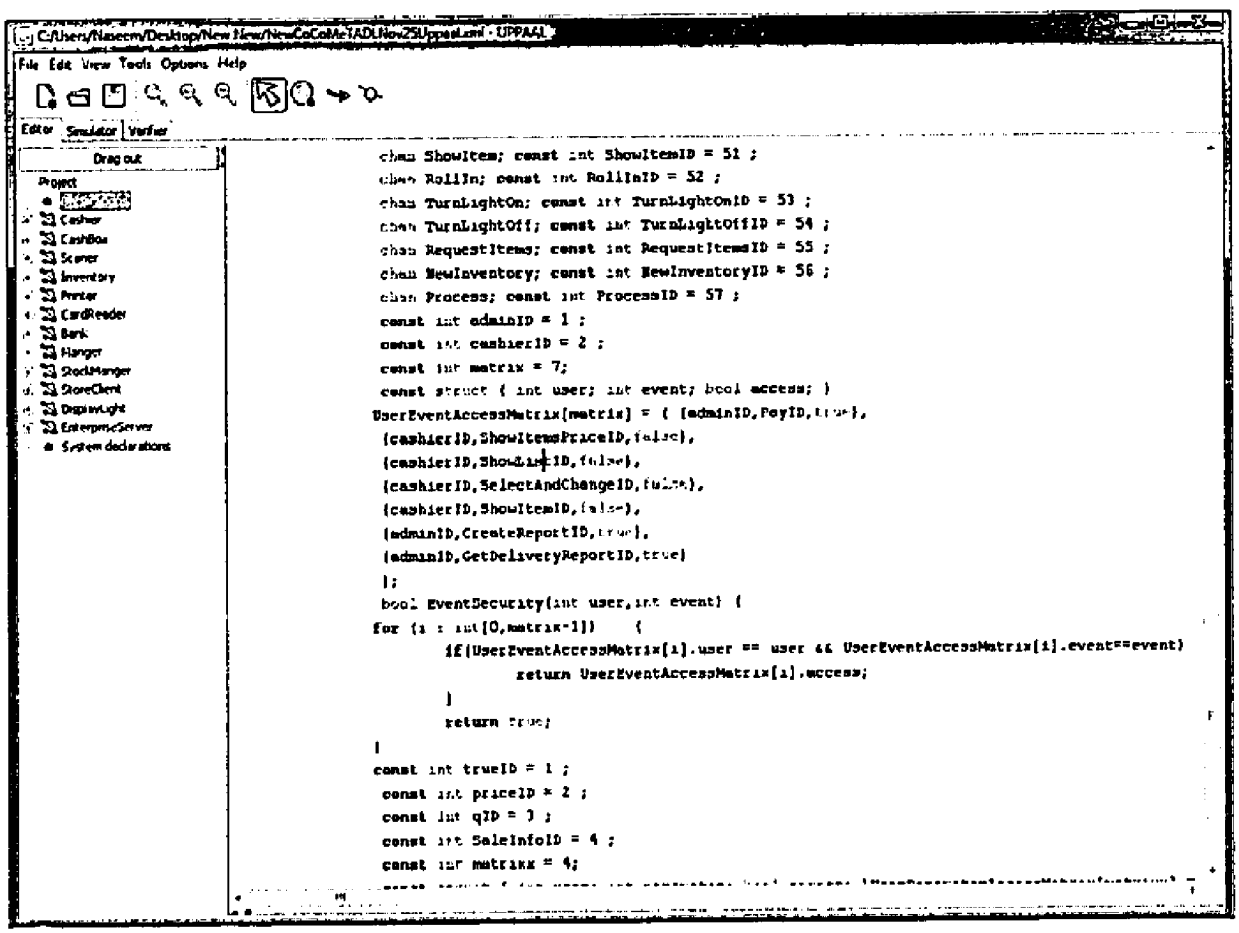

Figure 47: Global Declaration in UPPAAL

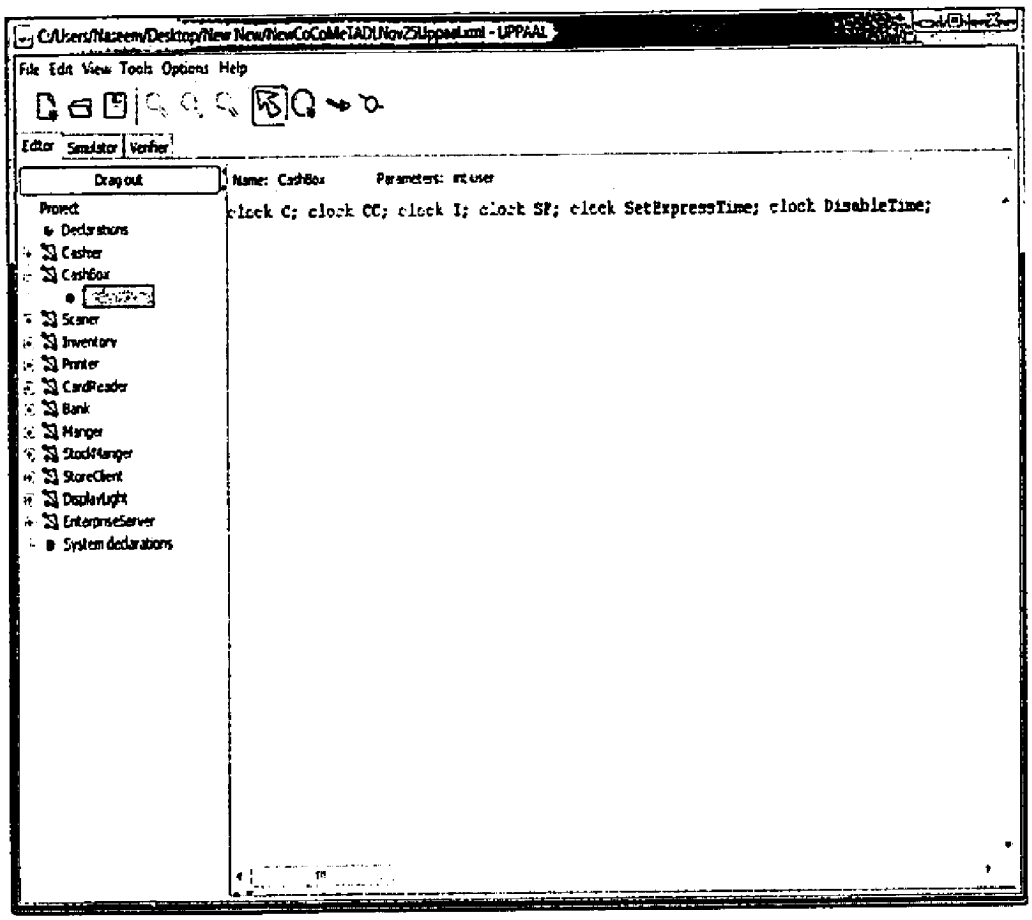

Figure 48: Cash Box Local Declaration in UPPAAL

TA representing this component, while Figure 48 contains the local-level declarations that were automatically generated, including the clock declarations. It can also be seen how the reactivities defined earlier were transformed into locations and transitions following the transformation rules discussed in Chapter 3.

Figures 50, 51, 52, 53, 54, 55 and 56 show the UPPAAL representation of the rest of the components. Figure 57 shows the system-level declaration that was generated automatically, and includes the instantiations of the templates by passing the *User* parameters.

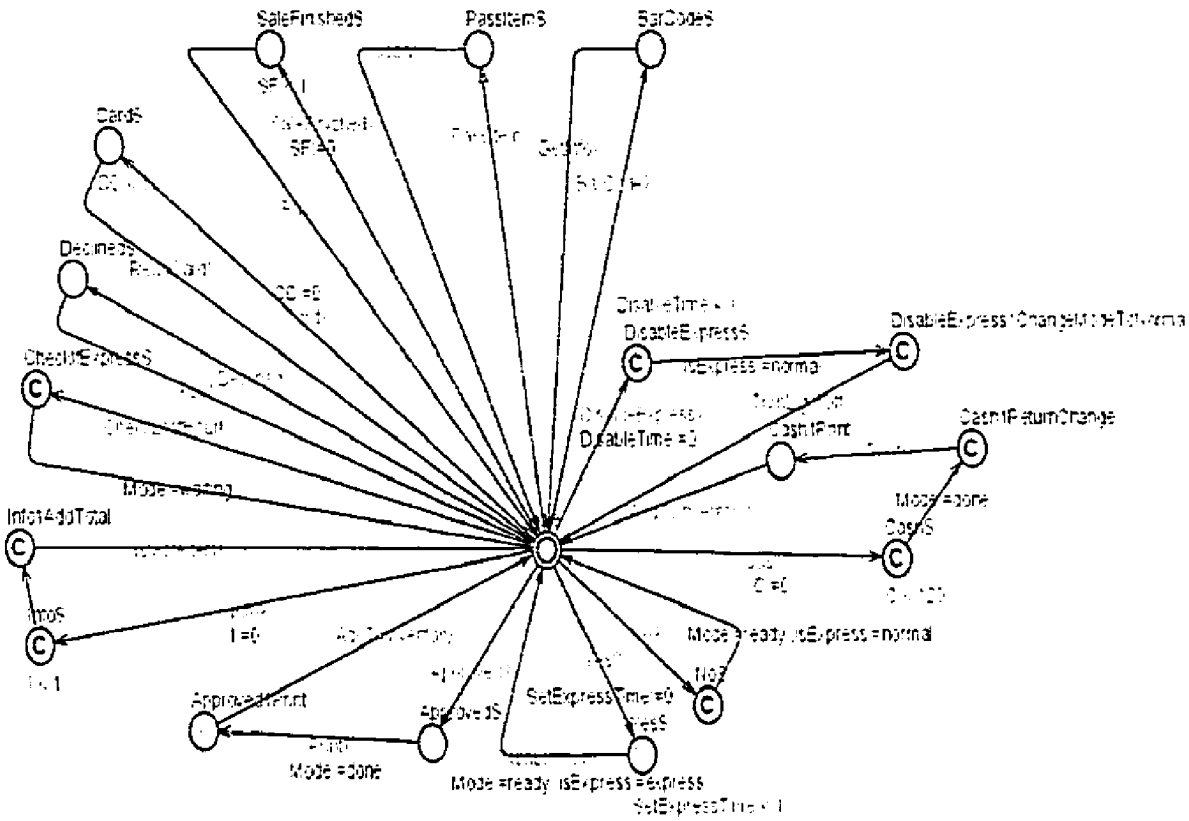

Figure 49: Cash Box template in UPPAAL

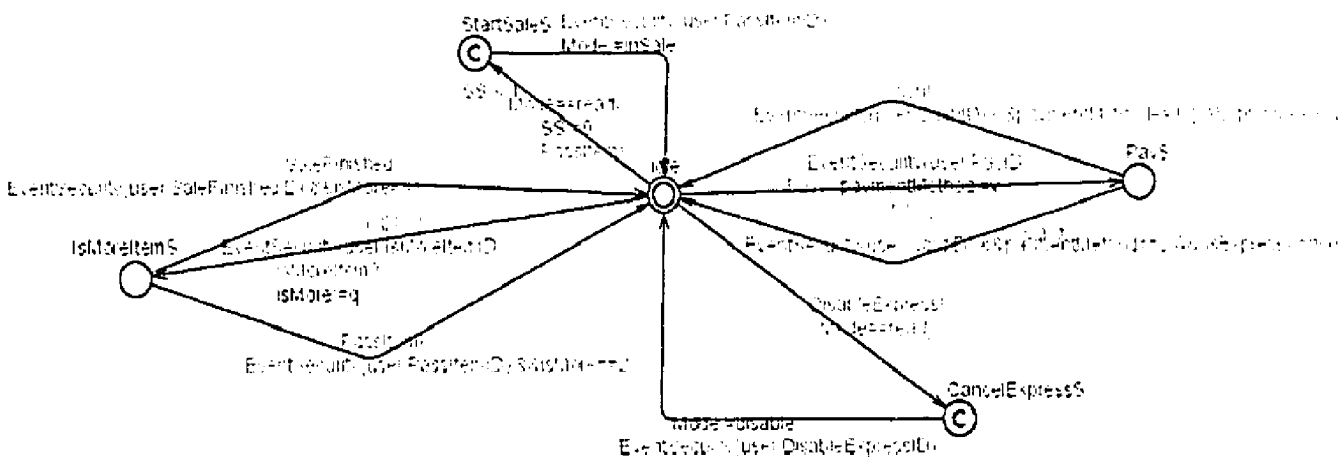

Figure 50: Cashier Template in UPPAAL

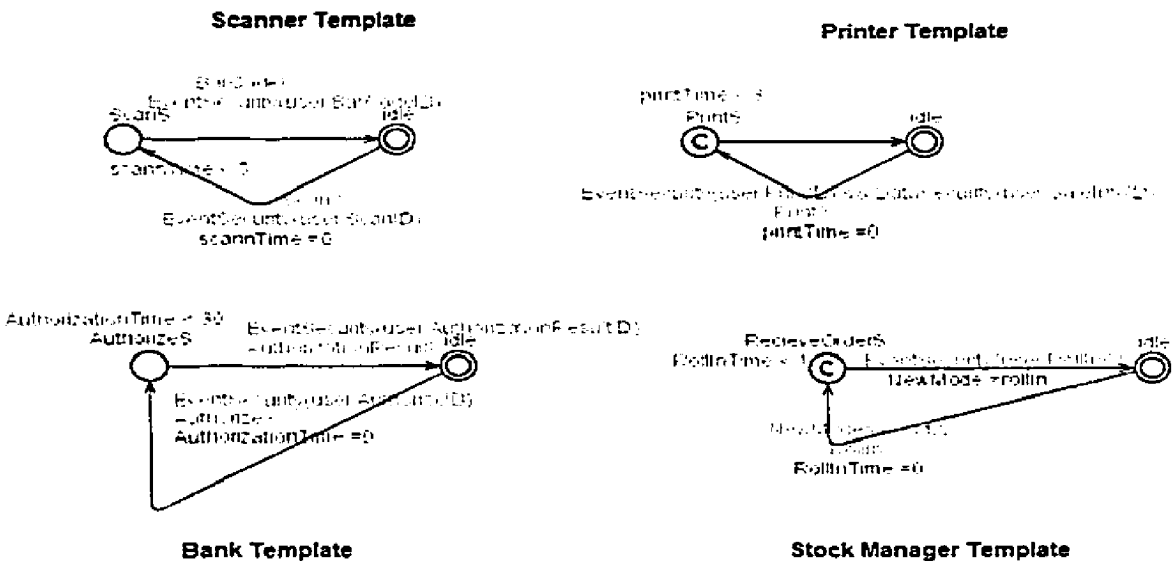

Figure 51: Scanner, Printer, Bank and Stock Manager Templates in UPPAAL

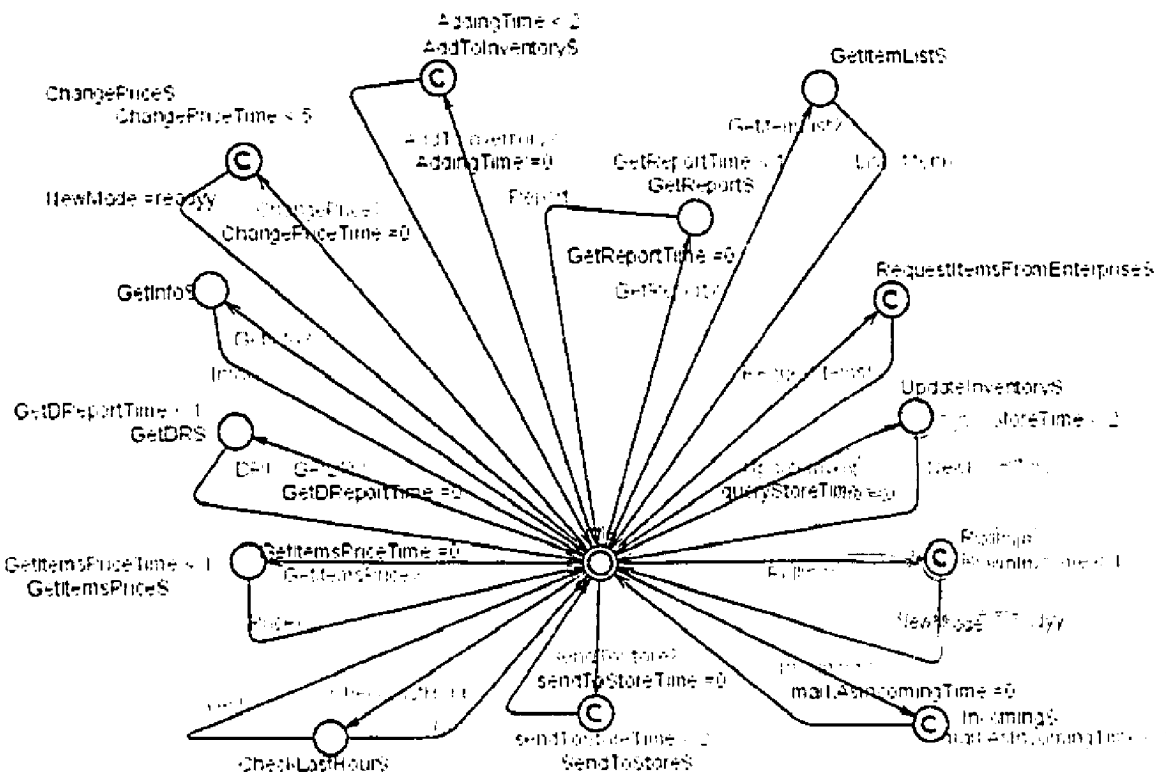

Figure 52: Inventory Template in UPPAAL

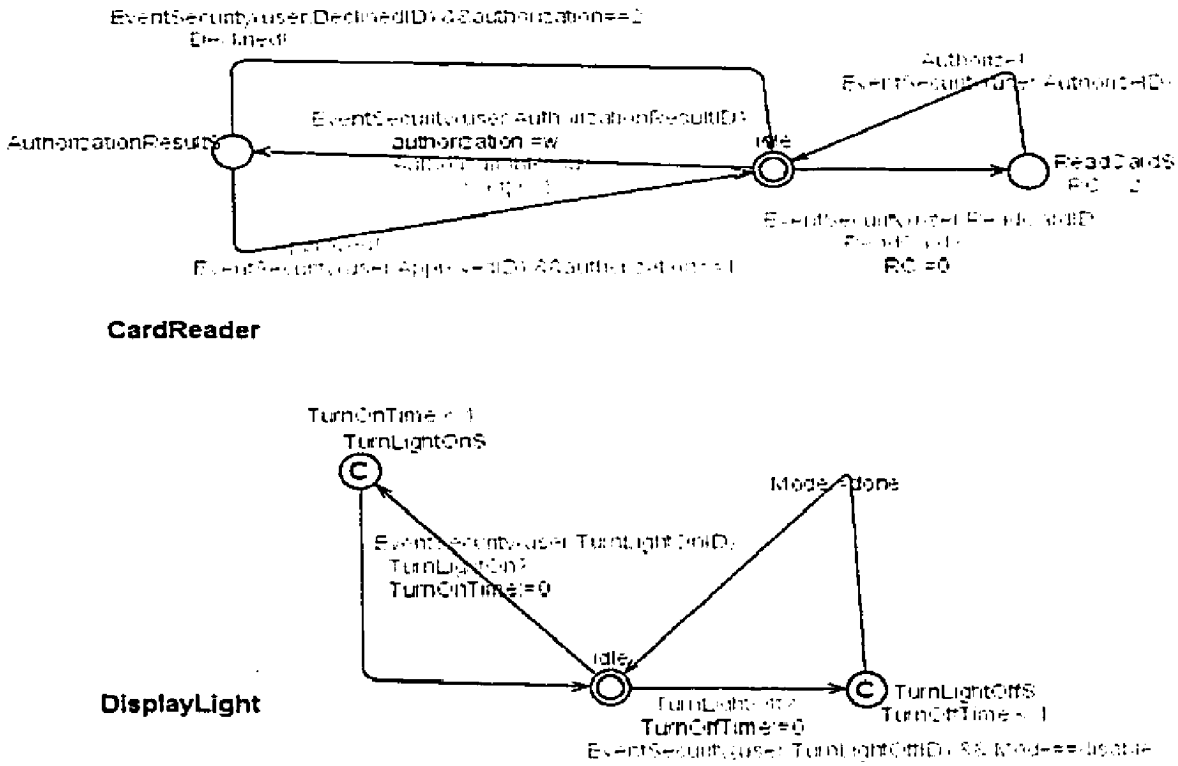

Figure 53: DisplayCard and CardReader Templates in UPPAAL

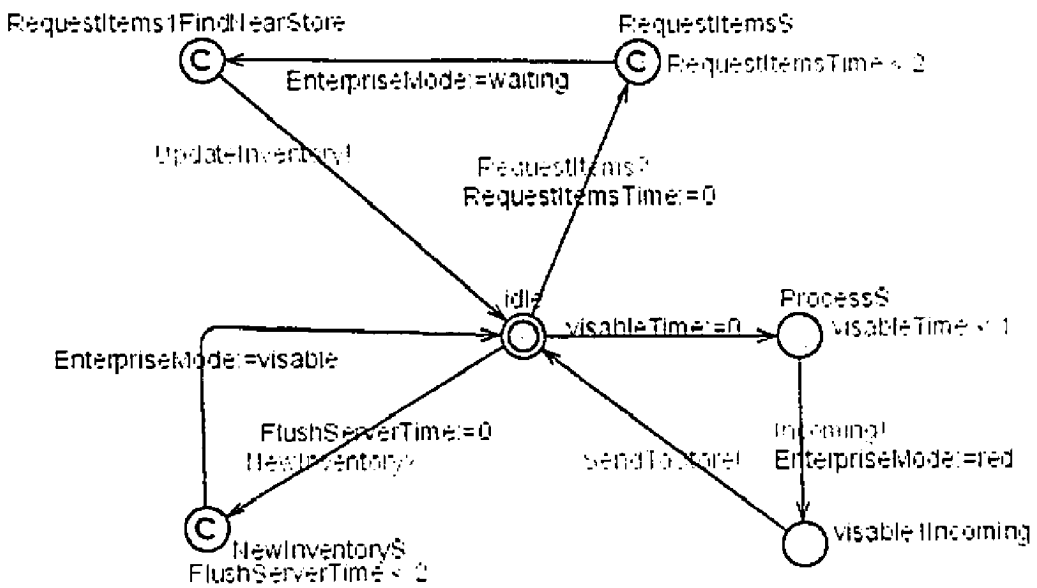

Figure 54: EnterpriseServer Template in UPPAAL

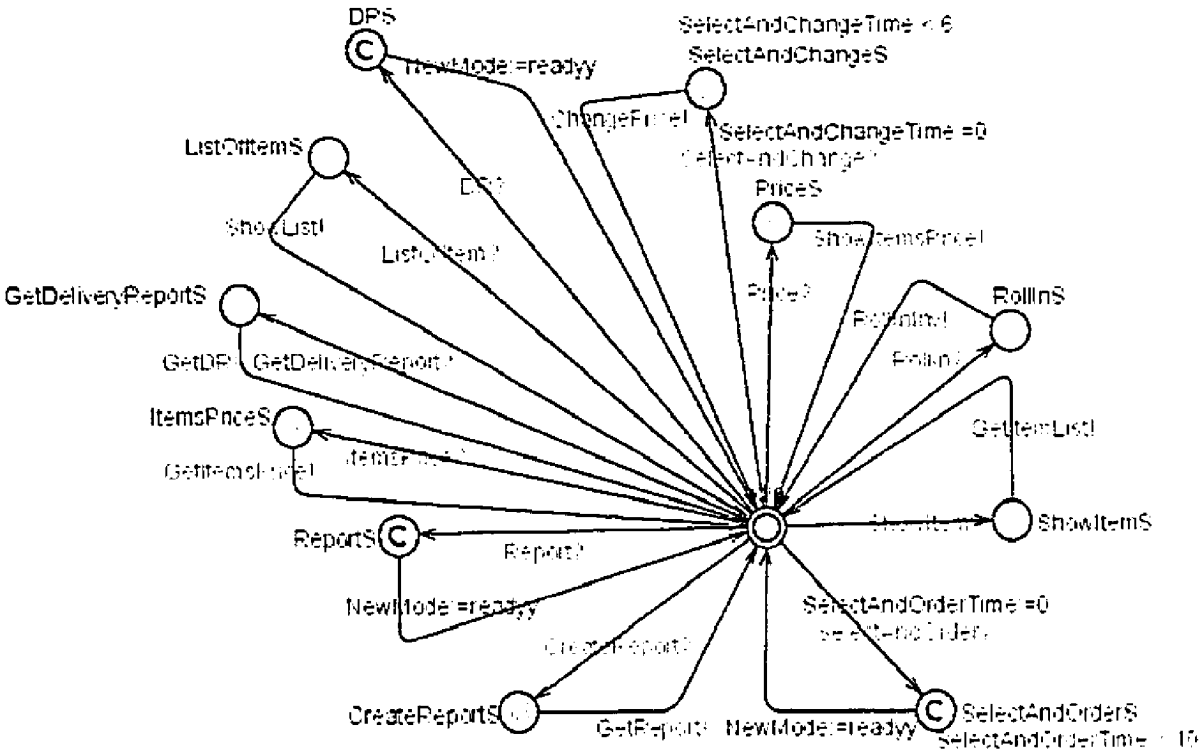

Figure 55: StoreClient Template in UPPAAL

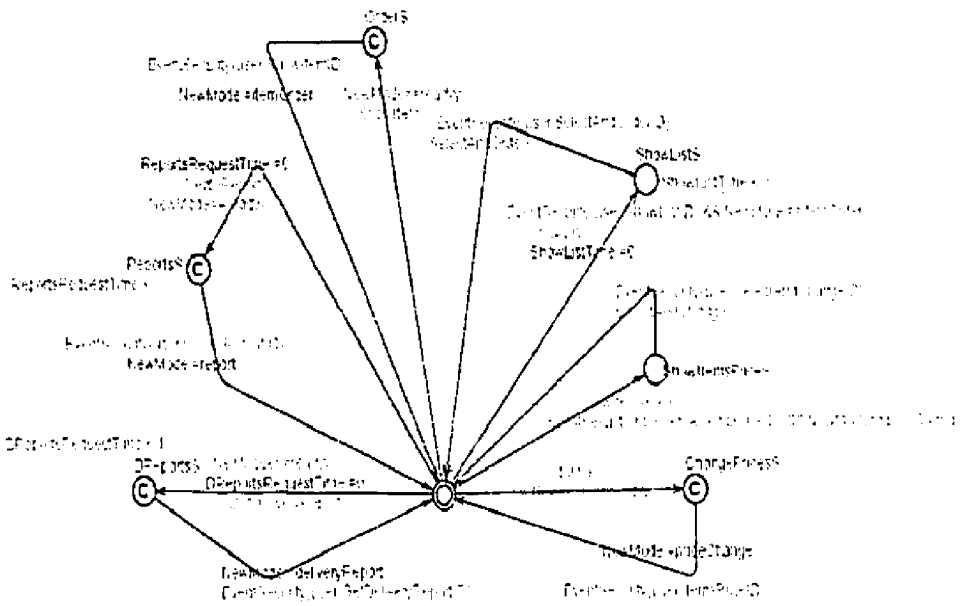

Figure 56: Manager Template in UPPAAL

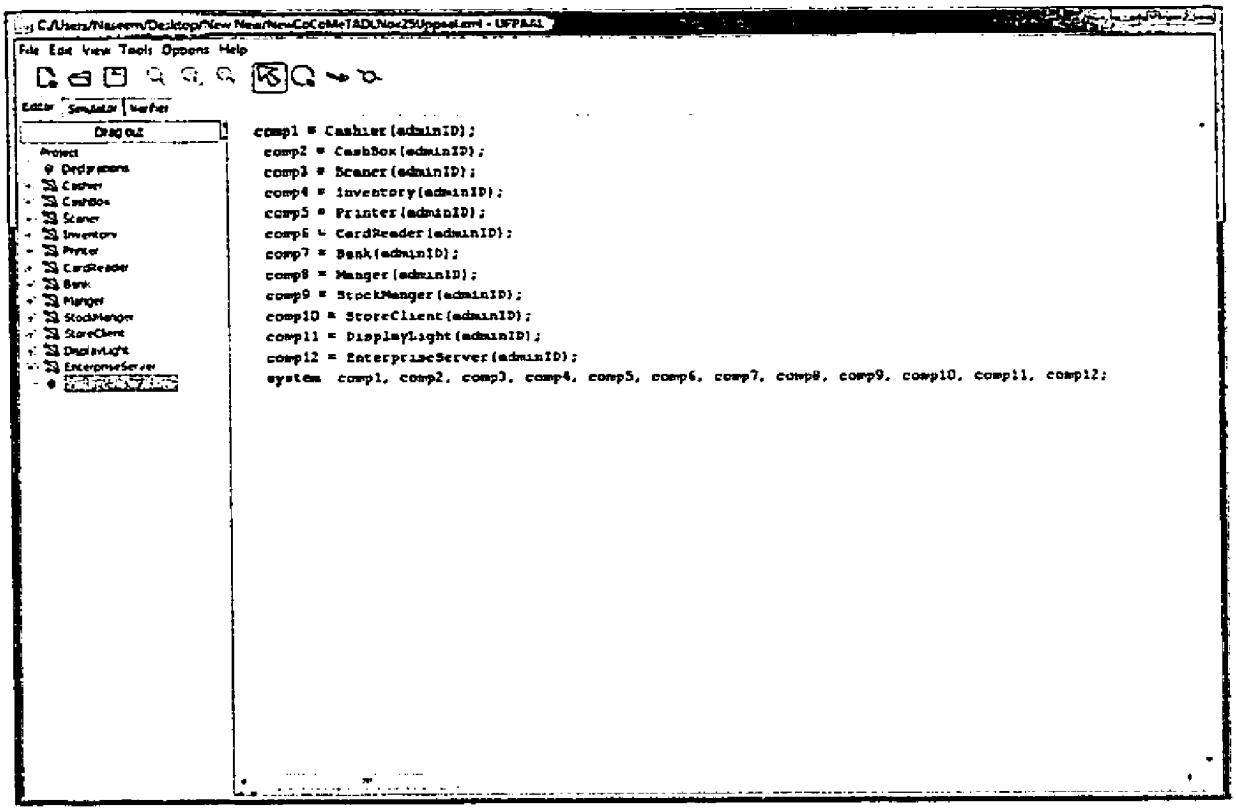

Figure 57: System Declaration in UPPAAL

Verification Rules

The UPPAAL verifier can be used to check the behaviour of the system by defining different checking formulas. This can be done manually by the user after the system has been transformed into the UPPAAL model and opened using the UPPAAL tool. TableTable 20 contains a sample of the safety and security properties that were tested on the resulting UPPAAL model of the CoCoME case study. It also contains a brief description of each property. Figure 58 shows how those properties were written within the UPPAAL tool and their corresponding verification results, which are displayed in the status bar.

Verification Rules	Notes
`A[] not deadlock`	Checks whether or not the system contains a deadlock
`E<> comp1.StartSaleS`	Means that there exists a way for the cashier to start a sale
`A[] comp2.CardS imply isExpress == normal`	Means that, if the system is processing a card payment, then the cash box can only be in the normal mode
`E<> comp1.CancelExpressS`	Means that there exists a way for the cashier to cancel the cash box's express mode
`A[] comp1.StartSaleS imply Mode==ready`	Means that, if the cashier is starting a sale, then the cash box should be in ready mode
`A[] comp2.CheckIfExpressS imply Mode==done`	Means that the cash box can only check whether or not it should transfer to express mode if the cash box is in done mode
`E<> comp8.OrderS`	Means that there exists a way for the manager to start an order process
`A[] comp8.OrderS imply NewMode == readyy`	Means that, if the manager is ordering new items, the store client is in ready mode
`E<> comp8.DReportsS`	Means that there exists a way for the manager to request delivery reports
`E<> comp8.ChangePricesS`	Means that there exists a way for the manager to start the change of price process
`E<> comp9.RecieveOrderS`	Means that the stock manager can start the receive order process
`comp2.CashS --> comp2.Cash1Print`	Means that a cash payment this will eventually result in the printing of a receipt
`comp2.InfoS --> comp2.Info1AddTotal`	Means that, if the cash box receives the item information, then it should update the sale total
`A[] comp12.RequestItemsS imply comp12.EnterpriseMode == comp12.red`	Means that the Enterprise Server can only process a request order in the ready mode
`A[] comp8.user == cashierID imply EventSecurity(cashierID, SelectAndChangeID)==false`	Means that the cashier cannot? change the price of items, can only be done by the manager

Table 20: Sample UPPAAL Safety and Security Properties

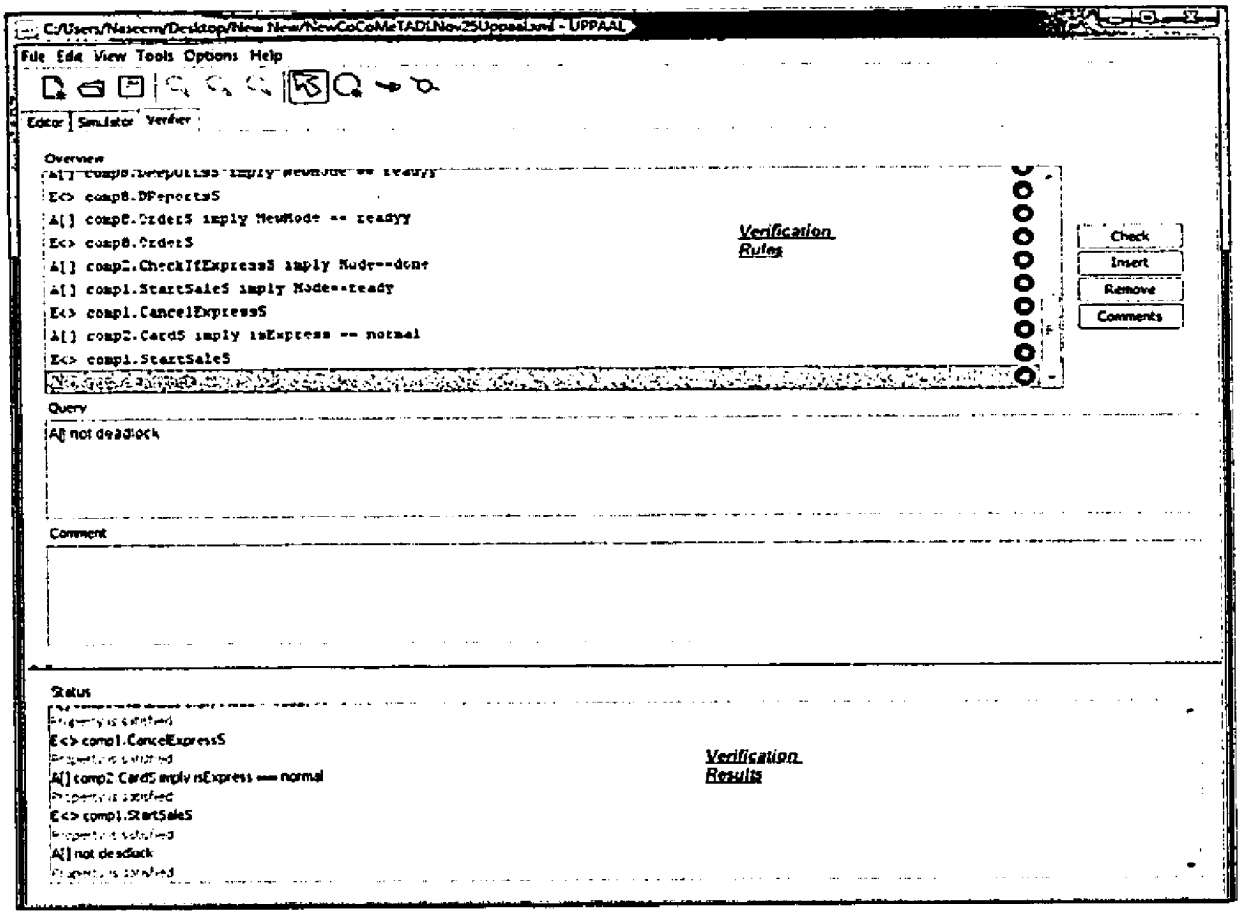

Figure 58: Verifying Safety and Security Properties in UPPAAL

5.2 Mine drainage

In this section, a simplified version of the Mine Drainage [BW90] case study is presented to illustrate the transformation process to the TIMES tool. This case study was chosen because it defines run-time requirements for the system.

5.2.1 Introduction

The example that has been chosen is based on one that commonly appears in the literature of real-time systems. It shows the software necessary to manage a simplified pump control system in a mining environment. The system is used to pump mine water, which collects in a sump at the bottom of the shaft, to the surface.

Task	Priority
Water Sensor	10
CO Sensor	7
Air Flow Sensor	5
CH4 Sensor	8

Table 21: Tasks Priorities

5.2.2 System overview

The system consists of two stations: one which controls the pump itself and one which monitors the environment in which the pump operates. The pump control station monitors the water levels at the sump. When the water reaches a high enough level, the pump turns on and the sump is drained until the water reaches the prescribed low level. The environment monitor detects the level of methane (CH4) in the air, the level of carbon monoxide (CO) in the mine, and whether or not there is an adequate flow of air. There are levels of CH4 and CO beyond which it is not safe to operate the pump.

5.2.3 System requirements

This section presents the functional and non-functional requirements of the Mine Drainage Control System. A more detailed presentation of the requirements can be found in [BW90]. Those requirements are:

- The CH4 sensor checks the CH4 level every 10 seconds.
- The CO sensor checks the CO level every 25 seconds.
- The air flow sensor checks the air flow every 60 seconds.
- The water flow sensor checks the water flow every 30 seconds.
- The tasks representing the processing of the system services have the priorities shown in Table 21.

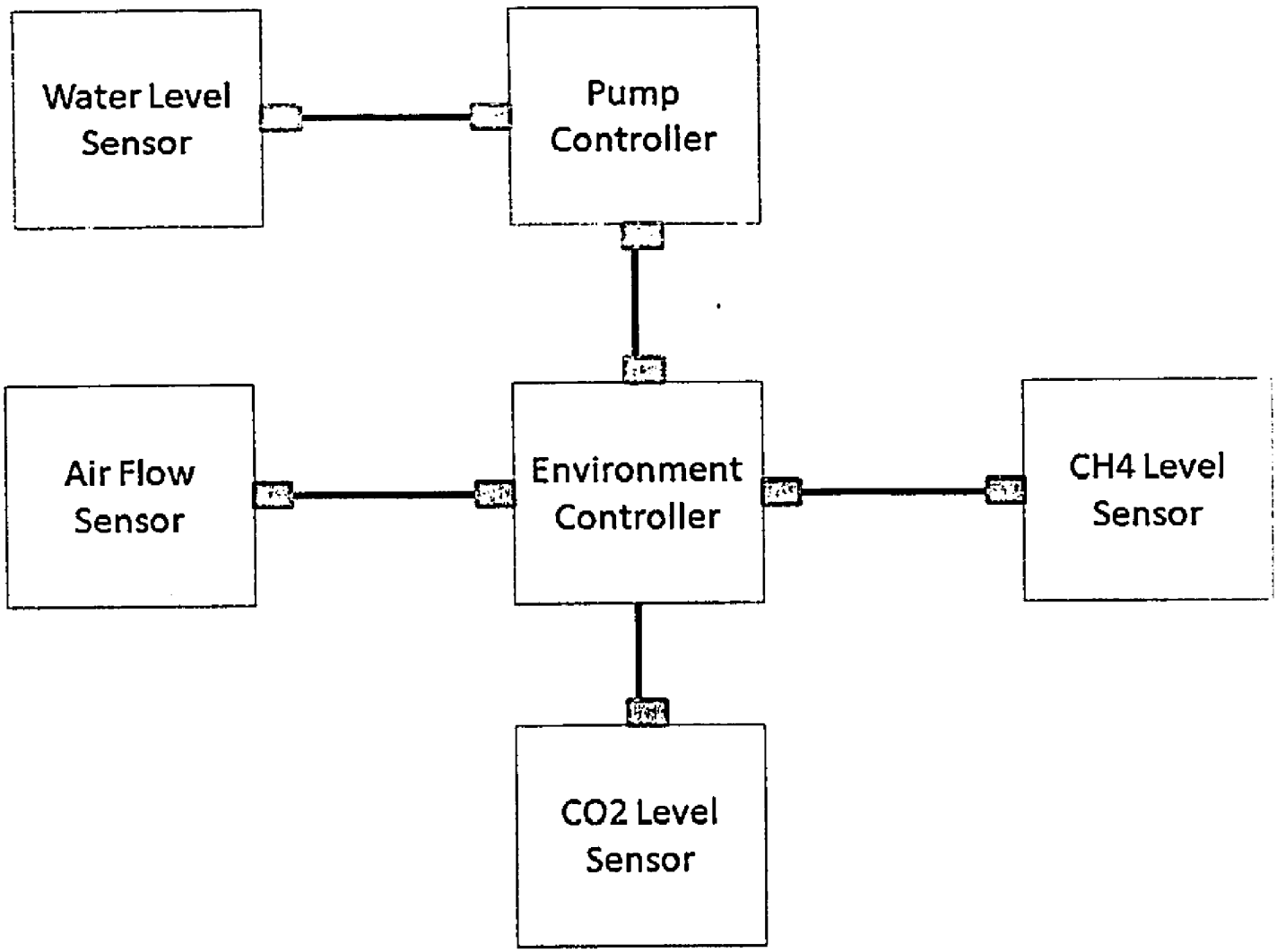

Figure 59: Mine Drainage Control System Components

5.2.4 TADL representation

The TADL representation starts by presenting the component diagram. From the previous description, it can be concluded that the system consists of six components, as depicted in Figure 59. Those components are: *Pump Controller, Environment Monitor, Water Level Sensor, CH4 Sensor, CO Sensor and Air Flow Sensor.* Before going into the details of each component, it is important to understand that, to be able to model the system, global-level variables are needed. These variables are introduced as system-level attributes and can be seen in Table 22. Following is a detailed description of the TADL representation of the above components. The concentration will be on the contract part of each component and

Variable	Type	Description
water	`int[1,10]`	Used to hold the water level.
co	`int`	Used to hold the CO level.
ch	`int`	Used to hold the CH4 level.
air	`int`	Used to hold the air flow level.

Table 22: System-level Variables

Request	Response	Data Constraint	Update
WaterLevel	isSafe	water<5	
WaterLevel	doNothing	water>=5 AND water <8	
WaterLevel	TurnPumpOff	water>=8	pump := off
safe	turnPumpOn		pump := on
notSafe	doNothing		

Table 23: PumpController Component Reactivities

the reactivities, as they are the only part included in the transformation process because they define component behaviour.

Pump Controller

This component is responsible for controlling the operation of the pump. It provides the following services:

- *Input Services :* WaterLevel, safe and notSafe.
- *Output Services :* isSafe.
- *Internal Services :* doNothing, turnPumpOff, turnPumpOn and doNothing.

This component has 5 reactivities, which can be seen in Table 23n

Water-level Sensor

This component is responsible for obtaining the water level. It provides two services: checkLevel (internal) and WaterLevel (output). The WaterLevel service represents a task that has the following properties: *Priority*, which is equal to 10, and *Period*, which is equal to 30 seconds.

Air Flow Sensor

This component is responsible for monitoring the Air Flow. It has two services: checkair (internal) and AirFlowSyn(output). The AirFlowSyn service represents a task with the following properties: *Priority*, which is equal to 5, and *Period*, which is equal to 60 seconds.

CH4 Sensor

This component is responsible for monitoring the CH4 level. It provides two services: checkch4 (internal) and CHSyn(output). The CHSyn service represents a task with the following properties: *Priority*, which is equal to 8, and *Period*, which is equal to 10 seconds.

CO Sensor

This component is responsible for monitoring the CO level. It provides two services: checkco (internal) and COSyn(output). The COSyn service represents a task with the following properties: *Priority*, which is equal to 7, and *Period*, which is equal to 25 seconds.

Environment Monitor

This component is responsible for monitoring the air flow, CH4 and CO sensors. It provides the following services:

- *Input Services :* isSafe, CHSyn, COSyn and AirFlowSyn.
- *Output Services :* safe and notSafe.
- *Internal Services :* doNothing.

This component has five reactivities, which can be seen in Table 24.

5.2.5 TIMES representation

The TADL XML representation of the Mine Drainage case study was defined in a single XML file. This file was passed to the TransformationTool, which produced the TIMES

Request	Response	Data Contraint
isSafe	safe	air==0 AND co==0 AND ch==0
isSafe	notSafe	air+co+ch != 0
CHSyn	doNothing	
COSyn	doNothing	
AirFlowSyn	doNothing	

Table 24: Environment Monitor Component Reactivities

representation of the system. The resulting XML file was opened in the TIMES tool. Below is a brief preview of the resulting TIMES system.

Figure 60 shows an overview of the TIMES tool after opening the resulted file from the transformation process. It shows the global-level declaration, which was automatically generated by the transformation process and contains the declaration of the global-level variables and the channels. The tasks definition can also be seen is Figure 60, which defines each task and its properties. These properties can be: Task Behaviour (B), Priority (P), Computing Time (C), Deadline (D) and Period (T). The system-level instantiation of the templates can also be seen in Figure 60.

Figure 61 represents the TIMES template of the PumpController component, while Figure 62 shows the TIME templates of the WaterSensor, AirFlowSensor, CH4Sensor, and CO2Sensor component. Finally, Figure 63 represents the TIMES templates of the EnvironmentMonitor component. The transitions and states that can be seen in each template correspond to the reactivities defined earlier and resulted by applying the transformation rules discussed in Chapter 3.

The TIMES tool is used to perform the schedulability analysis of the system tasks. Figure 64 shows the result of the schedulability analysis performed on the Mine Drainage case study.

5.3 Summary

In this chapter, two case studies have been presented. The first case study is the CoCoME case study which is used to illustrate the transformation to the UPPAAL model. The second

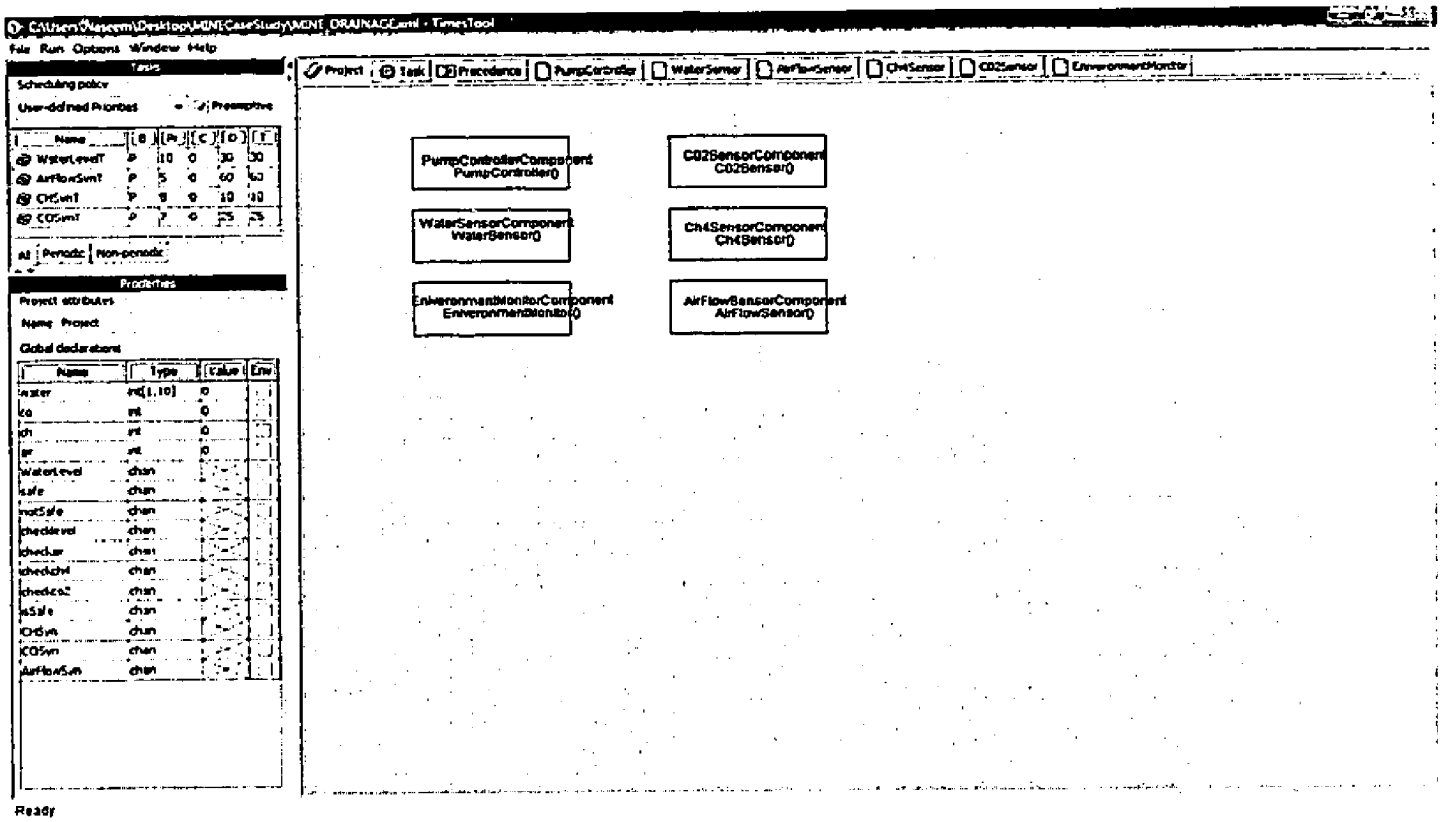

Figure 60: Mine Drainage Case Study in TIMES

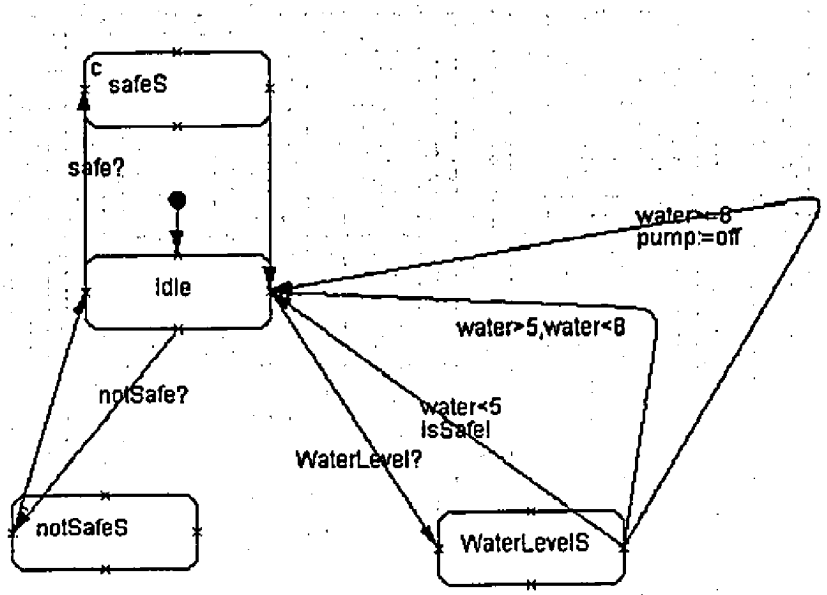

Figure 61: Pump Controller Template in TIMES

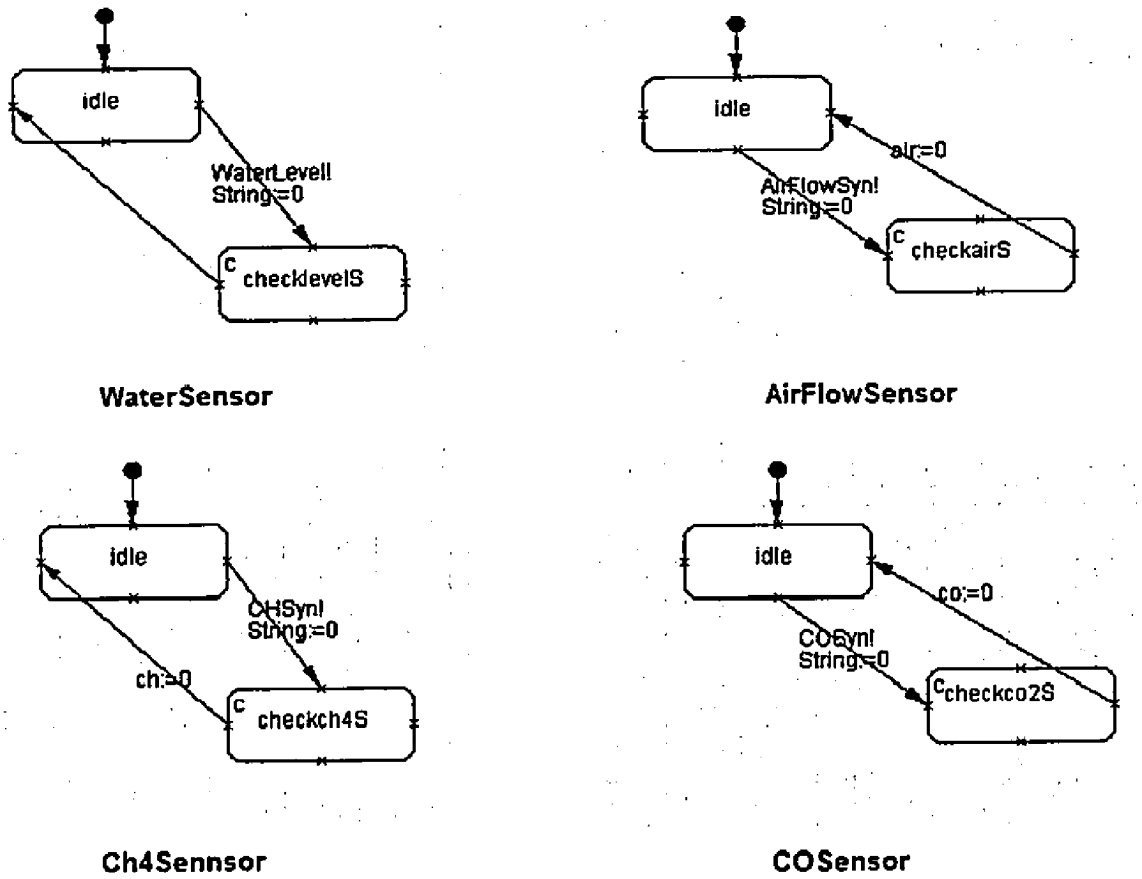

Figure 62: Water-, AirFlow-, CH4- and CO-Sensor Templates in TIMES

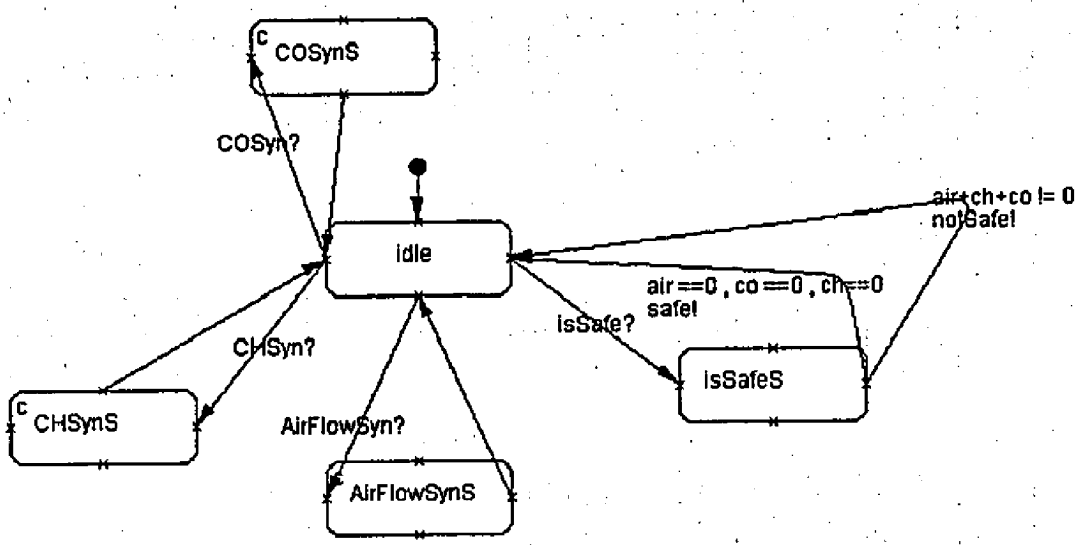

Figure 63: Environment Monitor Template in TIMES

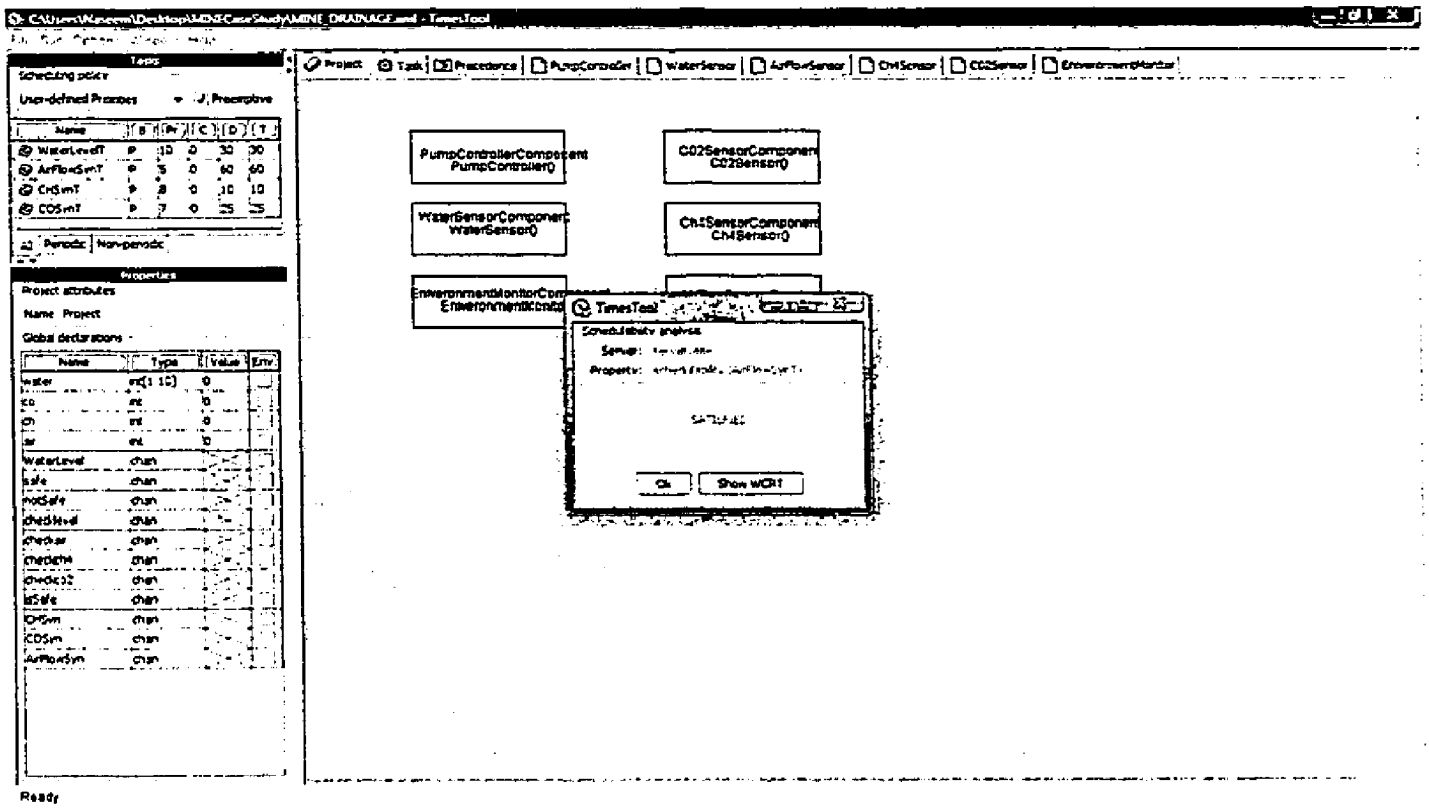

Figure 64: Schedulability Analysis Results in TIMES

case study is the Mine Drainage case study which is used to illustrate the transformation to the TIMES model.

Chapter 6

Conclusion

This thesis presents an automatic approach for analyzing ADL specification and generating behavioral models using model transformation techniques. The input for our transformation is TADL [MA08b], an architecture description language for trustworthy systems, and the output is UPPAAL [BDL04] extended timed automata or TIMES [AFM+02] extended time automata.

The significance of the approach presented here lies in its ability to separate the transformation rules from the transformation process, and utilize new technology to implement the transformation rules. This is very important, as it will increase the extensibility and maintainability of the model transformation. By defining the transformation rules separately, we will be able to change the input and output types by changing only the transformation rules, and not the transformation process.

In conventional approaches, the transformation rules are implemented using a series of *Loops* and *if..else* code in standard programming languages like Java or C#. The novelty of our approach is in utilizing new XML technology to implement the transformation rules using XSLT [Tid01]. XSLT has been very successful in representing the formal transformation rules, its most important advantage being its ability to formally define the transformation rules separately. Component models evolve with time, and new concepts can be added, removed or updated. Therefore, the transformation rules may require continuous updating, which can easily be achieved using XSLT. For example, in the early stages of our

work here, reactivity contained no update part. When this part was added, it became easy to modify the transformation rules written in XSLT to transform them into UPPAAL.

A tool was developed to automate the process of transformation from TADL XML to UPPAAL or TIMES XML. The tool uses the transformation rules that were defined in XSLT to perform the transformation.

Finally, this thesis presented the Common Component Modelling Example (CoCoME) case study, which was defined by the component development community to test the different component models, using TADL for the first time.

6.1 Future Work

6.1.1 Transformation rule and TransformationTool

In this thesis, the rules for transforming a system defined using TADL to the UPPAAL and TIMES models were defined. The same technique can be used to define the rules for transformation to other models, such as Promela. XSLT can also be used to define other transformation rules. The new transformation rules will enable us to verify the trustworthiness properties of the systems.

The TransformationTool defined in this thesis can also be easily extended to include new output models. This can be done by including only the transformation rules defined using XSLT. This is one of the main advantages of defining the transformation rules separately in XSLT.

6.1.2 UPPAAL transformation and TIMES transformation

UPPAAL tool gives a visual presentation of the extended TA as states and transitions. The resulting UPPAAL XML file from the transformation process presented those states and transitions without their coordinates on the screen (x and y). The coordinate is automatically given by the UPPAAL tool when it opens the file. This usually causes a problem in the case of complex templates. Currently, the user has to move the states or transitions

manually to increase the comprehensibility of the resulting view. A future improvement could be to introduce a new algorithm to automatically arrange the states and transitions in a clear way, which would eliminate the need for the user to manually arrange them.

The TIMES tool uses a different technique, in that it does not automatically give the coordinates to the containing elements, which means that the transformation process should add the coordinates to the transitions and states during that process. Currently, this is achieved by a very simple method, the user manually arranging the transitions, which usually generates a view that is not very clear. A future improvement could be to develop a more advanced algorithm to fix this problem.

6.1.3 XSLT and model transformation

The successful use of XSLT to perform model transformation opens the way for new uses of XSLT in the automatic transformation of component models. As XML is increasingly being used for model representation, the use of the transformation approach introduced in this thesis will increase as well.

Bibliography

[ABB+01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL - Now, Next, and Future. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Modelling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science Tutorial, pages 100–125. Springer–Verlag, 2001.

[ABD+01] M. Altheim, F. Boumphrey, S. Dooley, S. McCarron, S. Schnitzenrbaumer, and T. Wugofski. Modularization of xhtml. W3C recomendation report. Available at http://www.w3.org/TR/xhtml-modularization/, April 2001.

[AFM+02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times - a tool for modelling and implementation of embedded systems. In *8th International Conference, TACAS 2002, part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002*, Grenoble, France, April 2002.

[AFM+03] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *1st International Workshop on Formal Modeling and Analysis of Timed Systems, FORMATS 2003*, Marseille, France, September 2003.

[ALR01] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. Research report n01145, laas-cnrs, April 2001.

[AM07a] Vasu Alagar and Mubarak Mohammad. A component model for trustworthy real-time reactive systems development. In *International Workshop on Formal Aspects of Component Software (FACS07)*, Sophia-Antipolis, France, September 2007.

[AM07b] Vasu Alagar and Mubarak Mohammad. Specification and verification of trustworthy component-based real-time reactive systems. In *SAVCBS'07, Specification and Verification of Component-Based Systems*, Dubrovnik, Croatia, September 2007.

[Ama04] Xaviar Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing*. Phd thesis, Universitat Pompeu Fabra, October 2004.

[BD03] Brian Benz and John R. Durant. *XML Programming Bible*. Wiley Publishing, 2003.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[BW90] Alan Burns and Andy Wellings. *Real-Time Systems and Their PRogramming Languages*. Addison Wesley, 1990.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithims and tools. Report 316, The United Nation University, P.O.Box 305, Macau, September 2004.

[CL02] Ivica Crnkovic and Magnus Larsson, editors. *building reliable component-based Software Systems*. Artech House Publishers, 2002.

[DvdHT05] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Methodol.*, 14(2):199–245, 2005.

[GP02] Charles F. Goldfarb and Paul Prescod. *XML HANDBOOK*. Prentice Hall PTR, 4th edition, 2002.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[HKW+08] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. *The Common Component Modeling Example*, volume 5153 of *LNCS*, chapter CoCoME - The Common Component Modeling Example, pages 16–53. Springer, Heidelberg, 2008.

[Ibr08] Naseem Ibrahim. Naseem ibrahim home page, December 2008. http://users.encs.concordia.ca/~n_ibrah/.

[LBK98] Paul Clements Len Bass and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[MA08a] Mubarak Mohammad and Vasu Alagar. A framework for the development of trustworthy component-based systems. In *DSN'08, Submitted for review: IEEE/IFIP International Conference on Dependable Systems and Networks*, Alaska, USA, June 2008.

[MA08b] Mubarak Mohammad and Vasu Alagar. TADL - an architectural description language for trustworthy component-based systems. In *Proceedings of*

the 2nd European Conference of Software Architecture (ECSA'08), volume LNCS 5292, pages 290–297, Paphos, Cyprus, 2008. Springer-Verlag.

[MdVHC02] Craig Mundie, Pierre de Vries, Peter Haynes, and Matt Corwine. Trustworthy computing. Microsoft White Paper, October 2002.

[Pal04] Girish Keshav Palshikar. An introduction to model checking. *http://www.embedded.com/columns/technicalinsights/17603352?_requestid=179878*, December 2004.

[Pre01] Roger S. Pressman. *Software Engineering: A practitioner's approach.* McGraw Hill, fifth edition, 2001.

[SBI99] Fred B. Schneider, Steven M. Bellovin, and Alan S. Inouye. Building trustworthy systems: Lessons from the ptn and internet. *IEEE Internet Computing*, 3(6):64–72, 1999.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Prospectives on an emerging discipline.* Prentice Hall, 1996.

[Tid01] Doug Tidwell. *Mastering XML Transformation XSLT.* O'Reilly, 2001.

Appendix A

TADL XML schema

ComponenType schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:include schemaLocation=".\interfaceType.xsd"/>
    <xs:include schemaLocation=".\connectorType.xsd"/>
    <xs:include schemaLocation=".\contractType.xsd"/>
    <xs:include schemaLocation="RBAC.xsd"/>
    <xs:complexType name="ComponentType">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="property" type="Property" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="attribute" type="Attribute" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="constraint" type="xs:string" minOccurs="0"
                 maxOccurs="unbounded"/>
            <xs:element name="user" type="User" minOccurs="0"/>
            <xs:element name="interfaceTypes" type="InterfaceType"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="architectureType" type="ArchitectureType"
                minOccurs="0"/>
            <xs:element name="contract" type="ContractType" minOccurs="0
                "/>
            <xs:element name="discreption" type="xs:string" minOccurs="0
                "/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ArchitectureType">
        <xs:sequence>
```

```
            <xs:element name="name" type="xs:string"/>
            <xs:element name="componentType" type="ComponentType"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="connectorType" type="ConnectorType"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="attribute" type="Attribute" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="constraint" type="xs:string" minOccurs="0"
                 maxOccurs="unbounded"/>
            <xs:element name="attachments" type="Attachment" minOccurs="
                0" maxOccurs="unbounded"/>
            <xs:element name="descreption" type="xs:string" minOccurs="0
                "/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Attachment">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="connectorType-from" type="ConnectorType"/>
            <xs:element name="roleType-from" type="ConnectorRoleType"/>
            <xs:element name="interfaceType-from" type="InterfaceType"/>
            <xs:element name="componentType-to" type="ComponentType"/>
            <xs:element name="interfaceType-to" type="InterfaceType"/>
            <xs:element name="descreption" type="xs:string" minOccurs="0
                "/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

InterfaceType schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:complexType name="Property">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="value" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ServiceType">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="id" type="xs:string"/>
            <xs:element name="type" type="xs:string" minOccurs="0"/>
            <xs:element name="attribute" type="Attribute" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="constraint" type="xs:string" minOccurs="0"
                 maxOccurs="unbounded"/>
            <xs:element name="parameterType" type="ParameterType"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="proprty" type="Property" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="discreption" type="xs:string" minOccurs="0
                "/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ParameterType">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="datatype" type="xs:string"/>
            <xs:element name="value" minOccurs="0"/>
            <xs:element name="descreption" type="xs:string" minOccurs="0
                "/>
```

```
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Attribute">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="datatype" type="xs:string"/>
            <xs:element name="value" type="xs:string" minOccurs="0"/>
            <xs:element name="discription" type="xs:string" minOccurs="0
                "/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="InterfaceType">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="protocol" type="xs:string" minOccurs="0"/>
            <xs:element name="attribute" type="Attribute" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="serviceType" type="ServiceType" minOccurs=
                "0" maxOccurs="unbounded"/>
            <xs:element name="deiscreption" type="xs:string" minOccurs="
                0"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

ContractType schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:include schemaLocation=".\interfaceType.xsd"/>
    <xs:complexType name="SafetyProperty">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="serviceType" type="ServiceType" minOccurs=
                "0" maxOccurs="unbounded"/>
            <xs:element name="constraint" type="xs:string"/>
            <xs:element name="descreption" type="xs:string" minOccurs="0
                "/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ContractType">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dataConstraint" type="DataConstrain"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="timeConstraint" type="TimeConstrain"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="reactivity" type="Reactivity" minOccurs="0
                " maxOccurs="unbounded"/>
            <xs:element name="safetyProperty" type="SafetyProperty"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="descreption" type="xs:string" minOccurs="0
                "/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Reactivity">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="id" type="xs:string"/>
```

```
        <xs:element name="service-request" type="ServiceType"/>
        <xs:element name="service-response" type="ServiceType"/>
        <xs:element name="dataConstraint" type="DataConstrain"
            minOccurs="0"/>
        <xs:element name="timeConstraint" type="TimeConstrain"
            minOccurs="0"/>
        <xs:element name="update" type="Update" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element name="select" type="Select" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element name="action" minOccurs="0" maxOccurs="unbounded
            ">
            <xs:complexType>
                <xs:complexContent>
                    <xs:extension base="ServiceType">
                        <xs:sequence>
                            <xs:element name="from" type="xs:string"
                                />
                            <xs:element name="FromId" type="
                                xs:string"/>
                            <xs:element name="to" type="xs:string"
                                minOccurs="0"/>
                        </xs:sequence>
                    </xs:extension>
                </xs:complexContent>
            </xs:complexType>
        </xs:element>
        <xs:element name="descreption" type="xs:string" minOccurs="0
            "/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="TimeConstrain">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
```

```
        <xs:element name="attribute" type="Attribute" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element name="constraint" type="xs:string" minOccurs="0"
             maxOccurs="unbounded"/>
        <xs:element name="service-request" type="ServiceType"
            minOccurs="0"/>
        <xs:element name="service-resonse" type="ServiceType"
            minOccurs="0"/>
        <xs:element name="maxSafeTime" type="xs:int"/>
        <xs:element name="descreption" type="xs:string" minOccurs="0
            "/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="DataConstrain">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="service-request" type="ServiceType"/>
        <xs:element name="service-response" type="ServiceType"/>
        <xs:element name="constraint" type="xs:string"/>
        <xs:element name="descreption" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="Select">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="min" type="xs:string"/>
        <xs:element name="max" type="xs:string"/>
        <xs:element name="type" type="xs:string"/>
        <xs:element name="to" type="xs:string"/>
        <xs:element name="descreption" type="xs:string" minOccurs="0
            "/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="Update">
    <xs:sequence>
```

```
            <xs:element name="toBeUpdated" type="xs:string"/>
            <xs:element name="value" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

InterfaceType schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:include schemaLocation=".\componentType.xsd"/>
    <xs:include schemaLocation=".\RBAC.xsd"/>
    <xs:complexType name="SystemElement">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="descreption" type="xs:string" minOccurs="0
                "/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Deploy">
        <xs:sequence>
            <xs:element name="hardwareComponenetType" type="
                HardwareComponentType"/>
            <xs:element name="componentType" type="ComponentType"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="HardwareComponentType">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="attributes" type="Attribute" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="constraint" type="xs:string" minOccurs="0"
                />
            <xs:element name="interface" type="InterfaceType" minOccurs=
                "0" maxOccurs="unbounded"/>
            <xs:element name="descreption" type="xs:string" minOccurs="0
                "/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="Configuration">
```

```
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="attributes" type="Attribute" minOccurs
                    ="0" maxOccurs="unbounded"/>
                <xs:element name="components" type="ComponentType"
                    minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="deploy" type="Deploy" minOccurs="0"
                    maxOccurs="unbounded"/>
                <xs:element name="descreption" type="xs:string"
                    minOccurs="0"/>
                <xs:element name="rbac" type="RBAC" minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

ConnectorType schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:include schemaLocation=".\interfaceType.xsd"/>
    <xs:complexType name="ConnectorRoleType">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="attribute" type="Attribute" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="constraint" type="xs:string" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="interfaceType" type="InterfaceType"/>
            <xs:element name="descreption" type="xs:string" minOccurs="0
                "/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ConnectorType">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="connectorRoleType" type="ConnectorRoleType
                " minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="attribute" type="Attribute" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="constraint" type="xs:string" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="descreption" type="xs:string" minOccurs="0
                "/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

RBAC schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:include schemaLocation=".\interfaceType.xsd"/>
    <xs:complexType name="RBAC">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="users" type="User" minOccurs="0" maxOccurs
                ="unbounded"/>
            <xs:element name="groups" type="Group" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="roles" type="Role" minOccurs="0" maxOccurs
                ="unbounded"/>
            <xs:element name="privileges" type="Privilege" minOccurs="0"
                 maxOccurs="unbounded"/>
            <xs:element name="userGroupsAssignments" type="
                UserGroupAssignments" minOccurs="0" maxOccurs="unbounded
                "/>
            <xs:element name="userRolesAssignments" type="
                UserRolesAssignments" minOccurs="0" maxOccurs="unbounded
                "/>
            <xs:element name="groupRolesAssignments" type="
                GroupRolesAssignments" minOccurs="0" maxOccurs="
                unbounded"/>
            <xs:element name="serviceType" type="ServiceType" minOccurs=
                "0" maxOccurs="unbounded"/>
            <xs:element name="parameterType" type="ParameterType"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="privilegesForService" type="
                PrivilegesForServices" minOccurs="0" maxOccurs="
                unbounded"/>
```

```
        <xs:element name="privilegesForDataParameters" type="
            PrivilegesForDataParameters" minOccurs="0" maxOccurs="
            unbounded"/>
        <xs:element name="descreption" type="xs:string" minOccurs="0
            "/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="User">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="attribute" type="Attribute" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element name="contraint" type="xs:string" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element name="descreption" type="xs:string" minOccurs="0
            "/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="Group">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="attribute" type="Attribute" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element name="constraint" type="xs:string" minOccurs="0"
             maxOccurs="unbounded"/>
        <xs:element name="descreption" type="xs:string" minOccurs="0
            "/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="Role">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="attribute" type="Attribute" minOccurs="0"
            maxOccurs="unbounded"/>
```

```
        <xs:element name="contraint" type="xs:string" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element name="descreption" type="xs:string" minOccurs="0
            "/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="Privilege">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="attribute" type="Attribute" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element name="contraint" type="xs:string" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element name="descreption" type="xs:string" minOccurs="0
            "/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="UserGroupAssignments">
    <xs:sequence>
        <xs:element name="user" type="User"/>
        <xs:element name="group" type="Group"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="UserRolesAssignments">
    <xs:sequence>
        <xs:element name="user" type="User"/>
        <xs:element name="role" type="Role"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="GroupRolesAssignments">
    <xs:sequence>
        <xs:element name="group" type="Group"/>
        <xs:element name="role" type="Role"/>
    </xs:sequence>
</xs:complexType>
```

```
    <xs:complexType name="PrivilegesForServices">
        <xs:sequence>
            <xs:element name="service" type="ServiceType"/>
            <xs:element name="privilege" type="Privilege"/>
            <xs:element name="role" type="Role"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="PrivilegesForDataParameters">
        <xs:sequence>
            <xs:element name="dataParameter" type="ParameterType"/>
            <xs:element name="privilege" type="Privilege"/>
            <xs:element name="role" type="Role"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

PackageType schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:include schemaLocation="componentType.xsd"/>
    <xs:include schemaLocation="interfaceType.xsd"/>
    <xs:include schemaLocation="connectorType.xsd"/>
    <xs:complexType name="PackageType">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="Version"/>
            <xs:element name="interfaceTypes" type="InterfaceType"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="contractTypes" type="ContractType"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="connectorTypes" type="ConnectorType"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="componentTypes" type="ComponentType"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="descreption" type="xs:string" minOccurs="0
                "/>
            <xs:element name="packages" type="PackageType" minOccurs="0"
                 maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Appendix B

XSLT Transformation Rules

To UPPAAL Transformation rules

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/
    Transform" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:fn="http://www.w3.
    org/2005/xpath-functions" exclude-result-prefixes="fn xs xsi xsl">
  <xsl:output method="xml" encoding="UTF-8" indent="yes" doctype-system=
      ".\schemas\flat-1_0.dtd"/>
  <xsl:template match="/Configuration">
    <xsl:variable name="newline">
      <xsl:text>
    </xsl:text>
    </xsl:variable>
    <nta>
      <declaration>
        <xsl:for-each select="attributes">
          <xsl:variable name="n" select="name"/>
          <xsl:variable name="dt" select="datatype"/>
          <xsl:variable name="value" select="value"/>
          <xsl:sequence select="fn:concat($dt,' ',$n,' = ',$value,';',$
              newline)"/>
        </xsl:for-each>
        <xsl:sequence select="fn:concat('bool grant = true;',$newline)"/
            >
        <xsl:sequence select="$newline"/>
        <xsl:for-each-group select="components/contract/reactivity/
            service-request" group-by="name">
          <xsl:variable name="p" select="position()"/>
          <xsl:for-each select="current-group()[1]/name">
            <xsl:sequence select="fn:concat('chan ',xs:string(.),':
                const int ',xs:string(.),'ID = ',$p,' ;',$newline)"/>
```

```
  </xsl:for-each>
</xsl:for-each-group>
<xsl:variable name="stimCount" select="count(components/contract
    /reactivity/service-request)"/>
<xsl:for-each-group select="components/contract/reactivity/
    service-response" group-by="name">
  <xsl:variable name="p" select="position()"/>
  <xsl:for-each select="current-group()[1]/name">
  </xsl:for-each>
</xsl:for-each-group>
<xsl:variable name="responsCount" select="count(components/
    contract/reactivity/service-response)"/>
<xsl:for-each-group select="components/contract/reactivity/
    action" group-by="name">
  <xsl:variable name="p" select="position()"/>
  <xsl:for-each select="current-group()[1]/name">
  </xsl:for-each>
</xsl:for-each-group>
<xsl:for-each select="rbac/users/name">
  <xsl:sequence select="fn:concat('const␣int␣',xs:string(.),'ID␣
      =␣',position(),'␣;',$newline)"/>
</xsl:for-each>
<xsl:variable name="co" select="count(rbac/privilegesForService)
    "/>
<xsl:sequence select="fn:concat('const␣int␣matrix␣=␣',$co␣,';',$
    newline)"/>
<xsl:sequence select="xs:string('const␣struct␣{␣int␣user;␣int␣
    event;␣bool␣access;␣}UserEventAccessMatrix[matrix]␣=␣{')"/>
<xsl:for-each select="rbac/privilegesForService">
  <xsl:variable name="priv" select="privilege/name"/>
  <xsl:variable name="role" select="role/name"/>
  <xsl:variable name="service" select="service/name"/>
  <xsl:if test="$co␣!=␣position()">
    <xsl:sequence select="fn:concat('{',$role,'ID,',$service,'ID
        ,',$priv,'},',$newline)"/>
```

```
  </xsl:if>
  <xsl:if test="$co = position()">
    <xsl:sequence select="fn:concat('{',$role,'ID,',$service,'ID
        ,',$priv,'}',$newline)"/>
  </xsl:if>
</xsl:for-each>
<xsl:sequence select="fn:concat(xs:string('};'),$newline)"/>
<xsl:sequence select="xs:string('bool EventSecurity(int user,int
    event){   for(i: int[0,matrix-1])     {      if(
    UserEventAccessMatrix[i].user == user &amp;&amp;
    UserEventAccessMatrix[i].event==event)  return
    UserEventAccessMatrix[i].access;   } return true; }')"/>
<xsl:sequence select="$newline"/>
<xsl:for-each-group select="rbac/privilegesForDataParameters/
    dataParameter" group-by="name">
  <xsl:variable name="po" select="position()"/>
  <xsl:for-each select="current-group()[1]/name">
    <xsl:sequence select="fn:concat('const int ',xs:string(.),'
        ID = ',$po,' ;',$newline)"/>
  </xsl:for-each>
</xsl:for-each-group>
<xsl:variable name="coo" select="count(rbac/
    privilegesForDataParameters)"/>
<xsl:sequence select="fn:concat('const int matrixx = ',$coo
    ,';',$newline)"/>
<xsl:sequence select="xs:string('const struct { int user; int
    parameter; bool access; }UserParameterAccessMatrix[matrixx]
    ={')"/>
<xsl:for-each select="rbac/privilegesForDataParameters">
  <xsl:variable name="priv" select="privilege/name"/>
  <xsl:variable name="role" select="role/name"/>
  <xsl:variable name="data" select="dataParameter/name"/>
  <xsl:if test="$coo != position()">
    <xsl:sequence select="fn:concat('{',$role,'ID,',$data,'ID
        ,',$priv,'},',$newline)"/>
```

```
    </xsl:if>
    <xsl:if test="$coo = position()">
      <xsl:sequence select="fn:concat('{',$role,'ID,',$data,'ID
          ,',$priv,'}',$newline)"/>
    </xsl:if>
  </xsl:for-each>
  <xsl:sequence select="fn:concat(xs:string('};'),$newline)"/>
  <xsl:sequence select="xs:string('bool DataSecurity(int user, int
      parameter) {   for (i : int[0,1])    {      if(
      UserParameterAccessMatrix[i].user == user &amp;&amp;
      UserParameterAccessMatrix[i].parameter==parameter) return
      UserParameterAccessMatrix[i].access;    } return true; }')"/>
</declaration>
<xsl:for-each select="components">
  <template>
    <xsl:for-each select="name">
      <name>
        <xsl:sequence select="xs:string(.)"/>
      </name>
    </xsl:for-each>
    <parameter>int user</parameter>
    <declaration>
      <xsl:for-each-group select="contract/reactivity/
          timeConstraint" group-by="name">
        <xsl:for-each select="current-group()[1]/name">
          <xsl:sequence select="fn:concat('clock ',xs:string(.)
              ,';')"/>
        </xsl:for-each>
      </xsl:for-each-group>
      <xsl:for-each select="attribute">
        <xsl:variable name="name" select="name"/>
        <xsl:variable name="type" select="datatype"/>
        <xsl:variable name="value" select="value"/>
        <xsl:sequence select="fn:concat($type,' ',$name,' = ',$
            value,' ;')"/>
```

```
  </xsl:for-each>
</declaration>
<xsl:for-each-group select="contract/reactivity/service-
    request" group-by="name">
  <xsl:variable name="nuuu" select="count(current-group())"/>
  <xsl:for-each select="current-group()[1]/name">
    <location>
      <xsl:attribute name="id"><xsl:sequence select="xs:string
          (.)"/></xsl:attribute>
      <name>
        <xsl:sequence select="fn:concat(xs:string(.),'S')"/>
      </name>
      <xsl:variable name="vv" select="../../timeConstraint/
          name"/>
      <xsl:variable name="max" select="../../timeConstraint/
          maxSafeTime"/>
      <xsl:if test="../../timeConstraint">
        <label kind="invariant">
          <xsl:sequence select="fn:concat($vv,' &lt; ',$max)"/
              >
        </label>
      </xsl:if>
      <xsl:if test="$nuuu = 1">
        <xsl:if test="../../service-response/type = 'internal'
            ">
          <committed/>
        </xsl:if>
        <xsl:if test="../../service-request/type = 'internal'"
            >
          <xsl:if test="not(exists(../../action))">
          <committed/>
          </xsl:if>
        </xsl:if>
      </xsl:if>
    </location>
```

```
  </xsl:for-each>
</xsl:for-each-group>
<location>
  <xsl:attribute name="id"><xsl:sequence select="'idle'"/></
      xsl:attribute>
  <name>
    <xsl:sequence select="'idle'"/>
  </name>
</location>
<xsl:for-each select="contract/reactivity">
  <xsl:variable name="ren" select="."/>
  <xsl:if test="action">
    <location>
      <xsl:for-each select="service-response">
        <xsl:for-each select="name">
          <xsl:variable name="renn" select="$ren/name"/>
          <xsl:variable name="re" as="xs:string" select="
              fn:concat(xs:string($renn),␣xs:string(.))"/>
          <xsl:attribute name="id"><xsl:sequence select="$re"/
              ></xsl:attribute>
          <name>
            <xsl:sequence select="$re"/>
          </name>
        </xsl:for-each>
        <xsl:if test="type␣=␣'internal'">
        <committed/>
        </xsl:if>
      </xsl:for-each>


    </location>
  </xsl:if>
</xsl:for-each>
<xsl:for-each select="contract">
  <xsl:for-each select="reactivity">
```

```
    <xsl:variable name="reactivity-n" select="."/>
    <xsl:for-each select="action">
      <xsl:variable name="action-n" select="."/>
      <xsl:if test="not(exists(to))">
        <location>
          <xsl:for-each select="$reactivity-n/name">
            <xsl:variable name="reactivity-name" select="."/>
            <xsl:for-each select="$action-n/name">
              <xsl:variable name="result" as="xs:string"
                  select="fn:concat(xs:string($reactivity-name
                  ),␣xs:string(.))"/>
              <xsl:attribute name="id"><xsl:sequence select="$
                  result"/></xsl:attribute>
              <name>
                <xsl:sequence select="$result"/>
              </name>
            </xsl:for-each>
          </xsl:for-each>
          <xsl:if test="$reactivity-n/type␣=␣'internal'">
          <committed/>
          </xsl:if>

        </location>
      </xsl:if>
    </xsl:for-each>
  </xsl:for-each>
</xsl:for-each>
<init>
  <xsl:attribute name="ref"><xsl:sequence select="'idle'"/></
      xsl:attribute>
</init>
<xsl:for-each-group select="contract/reactivity/service-
    request" group-by="name">
  <xsl:variable name="nu" select="count(current-group())"/>
  <xsl:for-each select="current-group()[1]/name">
```

```
<transition>
  <source>
    <xsl:attribute name="ref"><xsl:sequence select="'idle'
        "/></xsl:attribute>
  </source>
  <target>
    <xsl:attribute name="ref"><xsl:sequence select="
        xs:string(.)"/></xsl:attribute>
  </target>
  <xsl:if test="not(../type = 'internal')">
    <label kind="synchronisation">
      <xsl:for-each select="../../service-request/name">
        <xsl:sequence select="fn:concat(xs:string(.),'?')"
            />
      </xsl:for-each>
    </label>
  </xsl:if>
  <xsl:if test="../type = 'internal'">
    <xsl:if test="$nu = 1">
      <xsl:if test="not(exists(../../action))">
      <label kind="synchronisation">
        <xsl:for-each select="../../service-response/name"
            >
          <xsl:sequence select="fn:concat(xs:string(.)
              ,'!')"/>
        </xsl:for-each>
      </label>
    </xsl:if>
    </xsl:if>
  </xsl:if>
  <xsl:if test="../../select">
    <xsl:for-each select="../../select">
      <xsl:variable name="na" select="name"/>
      <xsl:variable name="ty" select="type"/>
      <xsl:variable name="min" select="min"/>
```

```
<xsl:variable name="max" select="max"/>
<xsl:variable name="t" select="to"/>
<label kind="select">
  <xsl:sequence select="fn:concat($na,':',$ty,'[',$
      min,',',$max,']')"/>
</label>
<xsl:if test="../timeConstraint">
  <xsl:variable name="vv" select="../timeConstraint/
      name"/>
  <label kind="assignment">
    <xsl:sequence select="fn:concat($vv,':=0')"/>
    <xsl:sequence select="fn:concat(',',$t,':=',$na)
        "/>
    <xsl:if test="../type = 'internal'">
      <xsl:if test="$nu = 1">
        <xsl:if test="update">
          <xsl:sequence select="','"/>
          <xsl:for-each select="update">
            <xsl:variable name="toBeUpdated"
                select="toBeUpdated"/>
            <xsl:variable name="value" select="
                value"/>
            <xsl:if test="position() = 1">
              <xsl:sequence select="fn:concat($
                  toBeUpdated,':=',$value)"/>
            </xsl:if>
            <xsl:if test="position() != 1">
              <xsl:sequence select="fn:concat
                  (',',$toBeUpdated,':=',$value)"/
                  >
            </xsl:if>
          </xsl:for-each>
        </xsl:if>
      </xsl:if>
    </xsl:if>
```

```
      </label>
    </xsl:if>
    <xsl:if test="not(exists(../timeConstraint))">
      <xsl:variable name="vv" select="../timeConstraint/
          name"/>
      <label kind="assignment">
        <xsl:sequence select="fn:concat($t,':=',$na)"/>
      </label>
    </xsl:if>
  </xsl:for-each>
</xsl:if>
<xsl:if test="not(exists(../../select))">
  <xsl:if test="../../timeConstraint">
    <xsl:variable name="vv" select="../../timeConstraint
        /name"/>
    <label kind="assignment">
      <xsl:sequence select="fn:concat($vv,':=0')"/>
    </label>
  </xsl:if>
</xsl:if>
<label kind="guard">
  <xsl:if test="not(../type = 'internal')">
    <xsl:sequence select="fn:concat('EventSecurity(user
        ,',xs:string(.),'ID)')"/>
    <xsl:for-each select="../parameterType/name">
      <xsl:variable name="pamName" select="xs:string(.)"
          />
      <xsl:for-each select="/Configuration/rbac/
          privilegesForDataParameters/dataParameter/name
          ">
        <xsl:variable name="temp" select="xs:string(.)"/
            >
        <xsl:if test="$pamName = $temp">
          <xsl:sequence select="fn:concat('&amp;&amp;
              DataSecurity(user,',xs:string(.),'ID)')"/>
```

```
                </xsl:if>
              </xsl:for-each>
            </xsl:for-each>
          </xsl:if>
          <xsl:if test="$nu = 1">
            <xsl:if test="not(../type = 'internal')">
              <xsl:if test="../../dataConstraint">
                <xsl:for-each select="../../dataConstraint/
                    constraint">
                  <xsl:sequence select="fn:concat('&amp;amp;amp; ',
                      xs:string(.))"/>
                </xsl:for-each>
              </xsl:if>
            </xsl:if>
          </xsl:if>
          <xsl:if test="$nu = 1">
            <xsl:if test="../type = 'internal'">
              <xsl:if test="../../dataConstraint">
                <xsl:for-each select="../../dataConstraint/
                    constraint">
                  <xsl:sequence select="xs:string(.)"/>
                </xsl:for-each>
              </xsl:if>
            </xsl:if>
          </xsl:if>
        </label>
      </transition>
    </xsl:for-each>
  </xsl:for-each-group>
  <xsl:for-each select="contract/reactivity">
    <xsl:if test="action">
      <xsl:variable name="aa" select="name"/>
      <transition>
        <source>
          <xsl:for-each select="service-request/name">
```

```
    <xsl:attribute name="ref"><xsl:sequence select="
        xs:string(.)"/></xsl:attribute>
  </xsl:for-each>
</source>
<target>
  <xsl:for-each select="service-response/name">
    <xsl:variable as="xs:string" name="aaaa" select="
        fn:concat(xs:string($aa),␣xs:string(.))"/>
    <xsl:attribute name="ref"><xsl:sequence select="$
        aaaa"/></xsl:attribute>
  </xsl:for-each>
</target>
<xsl:if test="not(service-response/type␣=␣'internal')">
  <label kind="synchronisation">
    <xsl:for-each select="service-response/name">
      <xsl:sequence select="fn:concat(xs:string(.),'!')"
          />
    </xsl:for-each>
  </label>
</xsl:if>
<xsl:if test="update">
  <label kind="assignment">
    <xsl:for-each select="update">
      <xsl:variable name="toBeUpdated" select="
          toBeUpdated"/>
      <xsl:variable name="value" select="value"/>
      <xsl:if test="position()␣=␣1">
        <xsl:sequence select="fn:concat($toBeUpdated,':
            =',$value)"/>
      </xsl:if>
      <xsl:if test="position()␣!=␣1">
        <xsl:sequence select="fn:concat(',',$toBeUpdated
            ,':=',$value)"/>
      </xsl:if>
    </xsl:for-each>
```

```
  </label>
</xsl:if>
<label kind="guard">
  <xsl:if test="not(service-response/type = 'internal ')"
    >
    <xsl:for-each select="service-response/name">
      <xsl:sequence select="fn:concat('EventSecurity(
          user,',xs:string(.),'ID)')"/>
    </xsl:for-each>
    <xsl:for-each select="service-response/parameterType
      /name">
      <xsl:variable name="pamName" select="xs:string(.)"
        />
      <xsl:for-each select="/Configuration/rbac/
          privilegesForDataParameters/dataParameter/name
          ">
        <xsl:variable name="temp" select="xs:string(.)"/
          >
        <xsl:if test="$pamName = $temp">
          <xsl:sequence select="fn:concat('&amp;&amp; 
              DataSecurity(user,',xs:string(.),'ID)')"/>
        </xsl:if>
      </xsl:for-each>
    </xsl:for-each>
    <xsl:if test="dataConstraint">
      <xsl:for-each select="dataConstraint/constraint">
        <xsl:sequence select="fn:concat('&amp;&amp; ',
            xs:string(.))"/>
      </xsl:for-each>
    </xsl:if>
  </xsl:if>
  <xsl:if test="service-response/type = 'internal '">
    <xsl:for-each select="dataConstraint/constraint">
      <xsl:sequence select="xs:string(.)"/>
    </xsl:for-each>
```

```
      </xsl:if>
    </label>
    <xsl:if test="update">
      <label kind="assignment">
        <xsl:for-each select="update">
          <xsl:variable name="toBeUpdated" select="
              toBeUpdated"/>
          <xsl:variable name="value" select="value"/>
          <xsl:if test="position()=1">
            <xsl:sequence select="fn:concat($toBeUpdated,':
                =',$value)"/>
          </xsl:if>
          <xsl:if test="position()!=1">
            <xsl:sequence select="fn:concat(',',$toBeUpdated
                ,':=',$value)"/>
          </xsl:if>
        </xsl:for-each>
      </label>
    </xsl:if>
  </transition>
</xsl:if>
<xsl:if test="not(exists(action))">
  <transition>
    <source>
      <xsl:for-each select="service-request/name">
        <xsl:attribute name="ref"><xsl:sequence select="
            xs:string(.)"/></xsl:attribute>
      </xsl:for-each>
    </source>
    <target>
      <xsl:attribute name="ref"><xsl:sequence select="
          xs:string('idle')"/></xsl:attribute>
    </target>
    <xsl:if test="not(service-response/type='internal')">
      <xsl:if test="not(service-request/type='internal')">
```

```
    <label kind="synchronisation">
      <xsl:for-each select="service-response/name">
        <xsl:sequence select="fn:concat(xs:string(.)
            ,'!')"/>
      </xsl:for-each>
    </label>
  </xsl:if>
</xsl:if>
<xsl:variable name="kn" select="service-request/name"/>
<xsl:variable name="nu">
  <xsl:for-each-group select="../reactivity/service-
      request" group-by="name">
    <xsl:for-each select="current-group()[1]/name">
      <xsl:if test="xs:string(.) = $kn">
        <xsl:sequence select="count(current-group())"/>
      </xsl:if>
    </xsl:for-each>
  </xsl:for-each-group>
</xsl:variable>
<label kind="guard">
  <xsl:if test="not(service-response/type = 'internal')"
      >
    <xsl:for-each select="service-response/name">
      <xsl:sequence select="fn:concat('EventSecurity(
          user,',xs:string(.),'ID)')"/>
    </xsl:for-each>
    <xsl:for-each select="service-response/parameterType
        /name">
      <xsl:variable name="pamName" select="xs:string(.)"
          />
      <xsl:for-each select="/Configuration/rbac/
          privilegesForDataParameters/dataParameter/name
          ">
        <xsl:variable name="temp" select="xs:string(.)"/
            >
```

```
          <xsl:if test="$pamName = $temp">
            <xsl:sequence select="fn:concat('&amp;amp;&amp;amp;
                DataSecurity(user,',xs:string(.),'ID)')"/>
          </xsl:if>
        </xsl:for-each>
      </xsl:for-each>
      <xsl:if test="dataConstraint">
        <xsl:if test="$nu != 1">
          <xsl:for-each select="dataConstraint/constraint"
              >
            <xsl:sequence select="fn:concat('&amp;amp;&amp;amp;',
                xs:string(.))"/>
          </xsl:for-each>
        </xsl:if>
      </xsl:if>
    </xsl:if>
    <xsl:if test="service-response/type = 'internal'">
      <xsl:if test="dataConstraint">
        <xsl:if test="$nu != 1">
          <xsl:for-each select="dataConstraint/constraint"
              >
            <xsl:sequence select="xs:string(.)"/>
          </xsl:for-each>
        </xsl:if>
      </xsl:if>
    </xsl:if>
  </label>
  <xsl:if test="update">
    <label kind="assignment">
      <xsl:for-each select="update">
        <xsl:variable name="toBeUpdated" select="
            toBeUpdated"/>
        <xsl:variable name="value" select="value"/>
        <xsl:if test="position() = 1">
```

```
                <xsl:sequence select="fn:concat($toBeUpdated,':
                    =',$value)"/>
              </xsl:if>
              <xsl:if test="position()!=1">
                <xsl:sequence select="fn:concat(',',$toBeUpdated
                    ,':=',$value)"/>
              </xsl:if>
            </xsl:for-each>
          </label>
        </xsl:if>
      </transition>
    </xsl:if>
  </xsl:for-each>
  <xsl:for-each select="contract/reactivity">
    <xsl:variable name="react-name" select="name"/>
    <xsl:for-each select="action">
      <xsl:if test="not(exists(to))">
        <transition>
          <source>
            <xsl:for-each select="from">
              <xsl:variable name="reso" as="xs:string" select="
                  fn:concat(xs:string($react-name),'xs:string(.)
                  )"/>
              <xsl:attribute name="ref"><xsl:sequence select="$
                  reso"/></xsl:attribute>
            </xsl:for-each>
          </source>
          <target>
            <xsl:for-each select="name">
              <xsl:variable name="res" as="xs:string" select="
                  fn:concat(xs:string($react-name),'xs:string(.)
                  )"/>
              <xsl:attribute name="ref"><xsl:sequence select="$
                  res"/></xsl:attribute>
            </xsl:for-each>
```

```
</target>
<xsl:if test="not(type = 'internal')">
  <label kind="synchronisation">
    <xsl:for-each select="name">
      <xsl:sequence select="fn:concat(xs:string(.)
          ,'!')"/>
    </xsl:for-each>
  </label>
</xsl:if>
<label kind="guard">
  <xsl:if test="not(type = 'internal')">
    <xsl:for-each select="name">
      <xsl:sequence select="fn:concat('EventSecurity(
          user,',xs:string(.),'ID)')"/>
    </xsl:for-each>


    <xsl:for-each select="parameterType/name">
    <xsl:variable name="pamName" select="xs:string(.)"
        />
    <xsl:for-each select="/Configuration/rbac/
        privilegesForDataParameters/dataParameter/name
        ">
      <xsl:variable name="temp" select="xs:string(.)"/
          >
      <xsl:if test="$pamName = $temp">
        <xsl:sequence select="fn:concat('&amp;amp;amp; 
            DataSecurity(user,',xs:string(.),'ID)')"/>
      </xsl:if>
    </xsl:for-each>
  </xsl:for-each>

  </xsl:if>
 </label>
</transition>
```

```
</xsl:if>
<xsl:if test="to">
  <transition>
    <source>
      <xsl:for-each select="from">
        <xsl:variable name="resss" as="xs:string" select="
            fn:concat(xs:string($react-name),␣xs:string(.)
            )"/>
        <xsl:attribute name="ref"><xsl:sequence select="$
            resss"/></xsl:attribute>
      </xsl:for-each>
    </source>
    <target>
      <xsl:attribute name="ref"><xsl:sequence select="'
          idle'"/></xsl:attribute>
    </target>
    <xsl:if test="not(type␣=␣'internal')">
      <label kind="synchronisation">
        <xsl:for-each select="name">
          <xsl:sequence select="fn:concat(xs:string(.)
              ,'!')"/>
        </xsl:for-each>
      </label>
    </xsl:if>
    <lab l kind="guard">
      <xsl:if test="not(type␣=␣'internal')">
        <xsl:for-each select="name">
          <xsl:sequence select="fn:concat('EventSecurity(
              user,',xs:string(.)␣,'ID)')"/>
        </xsl:for-each>
        <xsl:for-each select="parameterType/name">
          <xsl:variable name="pamName" select="xs:string
              (.)"/>
```

```
                    <xsl:for-each select="/Configuration/rbac/
                        privilegesForDataParameters/dataParameter/
                        name">
                      <xsl:variable name="temp" select="xs:string(.)
                        "/>
                      <xsl:if test="$pamName = $temp">
                        <xsl:sequence select="fn:concat('&amp;amp;&amp;amp;
                          DataSecurity(user,',xs:string(.),'ID)')"
                          />
                      </xsl:if>
                    </xsl:for-each>
                  </xsl:for-each>
                </xsl:if>
              </label>
            </transition>
          </xsl:if>
        </xsl:for-each>
      </xsl:for-each>
    </template>
  </xsl:for-each>
  <system>
    <xsl:for-each select="components">
      <xsl:sequence select="fn:concat('comp',position(),' = ',
          xs:string(name),'(adminID);',$newline)"/>
    </xsl:for-each>
    <xsl:sequence select="xs:string('system ')"/>
    <xsl:variable name="c" select="count(components)"/>
    <xsl:for-each select="components">
      <xsl:if test="position() != $c">
        <xsl:sequence select="fn:concat('comp',position(),',')"/>
      </xsl:if>
      <xsl:if test="position() = $c">
        <xsl:sequence select="fn:concat('comp',position(),';')"/>
      </xsl:if>
    </xsl:for-each>
```

```
        </system>
      </nta>
    </xsl:template>
</xsl:stylesheet>
```

To TIMES Transformation rules

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/
    Transform" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:fn="http://www.w3.
    org/2005/xpath-functions"
                exclude-result-prefixes="fn xs xsi xsl">
  <xsl:output method="xml" encoding="UTF-8" indent="yes"/>
  <xsl:template match="/Configuration">
    <xsl:variable name="newline">
      <xsl:text>
    </xsl:text>
    </xsl:variable>
    <times>
      <system>
        <name>Project</name>

        <declarations>
          <xsl:for-each select="attributes">
            <xsl:variable name="n" select="name"/>
            <xsl:variable name="dt" select="datatype"/>
            <xsl:variable name="value" select="value"/>
            <declaration name="{$n}" type="{$dt}" value="{$value}"/>
          </xsl:for-each>

          <xsl:for-each-group select="components/contract/reactivity/
              service-request" group-by="name">
            <xsl:variable name="p" select="position()"/>
            <xsl:for-each select="current-group()[1]/name">
              <declaration name="{xs:string(.)}" type="chan" value="0"/>
            </xsl:for-each>
          </xsl:for-each-group>
        </declarations>
        <xsl:for-each select="components">
```

```
    <process>
      <portcontainer>
        <nailcontainer>
          <labelcontainer>
            <object>
              <metrics h="50" w="140" x="80" y="50"/>
              <misc color="b6d2c2" id="0"/>
            </object>
            <xsl:for-each select="name">
              <label text="{fn:concat(xs:string(.),'Component')}"
                  x="92" y="62"/>
            </xsl:for-each>
            <font name="SansSerif"/>
          </labelcontainer>
        </nailcontainer>
      </portcontainer>
      <xsl:for-each select="name">
        <misc template="{xs:string(.)}"/>
      </xsl:for-each>
    </process>
  </xsl:for-each>
</system>

<tasktable schedulingpolicy="EDF">
  <xsl:for-each select="components/contract/reactivity">
    <xsl:for-each select="service-response">
      <task>
        <xsl:for-each select="property">
          <xsl:attribute name="name" select="value"/>
        </xsl:for-each>
        <xsl:attribute name="name" select="fn:concat(name,'T')"/>
        <xsl:if test="../timeConstraint">
          <xsl:attribute name="D" select="../timeConstraint/
              maxSafeTime"/>
        </xsl:if>
```

```
      </task>
    </xsl:for-each>
  </xsl:for-each>
</tasktable>
<xsl:for-each select="components">
  <template>
    <xsl:for-each select="property">
      <xsl:if test="name = 'environment'">
        <xsl:attribute name="environment" select="'true'"/>
      </xsl:if>
    </xsl:for-each>
    <xsl:for-each select="name">
      <name>
        <xsl:sequence select="xs:string(.)"/>
      </name>
    </xsl:for-each>

    <declarations>

      <xsl:for-each-group select="contract/reactivity/
          timeConstraint" group-by="name">
        <xsl:for-each select="current-group()[1]/name">
          <declaration name="{xs:string(.)}" type="clock" value="0
              "/>
        </xsl:for-each>
      </xsl:for-each-group>

      <xsl:for-each select="attribute">
        <xsl:variable name="name" select="name"/>
        <xsl:variable name="type" select="datatype"/>
        <xsl:variable name="value" select="value"/>
        <declaration name="{$name}" type="{$type}" value="{$value}
            "/>
      </xsl:for-each>
    </declarations>
```

```
<xsl:for-each-group select="contract/reactivity/service-
    request" group-by="name">
  <xsl:variable name="nuuu" select="count(current-group())"/>
  <xsl:variable name="i" select="current-group()[1]/id"/>
  <xsl:for-each select="current-group()[1]/name">
    <location>
      <portcontainer>
        <nailcontainer>
          <labelcontainer>
            <object>
              <metrics h="40" w="100" x="340" y="140"/>
              <misc id="{$i}"/>
            </object>
            <label text="{fn:concat(xs:string(.),'S')}" x="352
                " y="150"/>
            <font name="SansSerif"/>
          </labelcontainer>
        </nailcontainer>
      </portcontainer>
      <xsl:variable name="vv" select="../../timeConstraint/
          name"/>
      <xsl:variable name="max" select="../../timeConstraint/
          maxSafeTime"/>
      <xsl:if test="../../timeConstraint">
        <misc invariant="{fn:concat($vv,' &lt; ',$max)}"/>
      </xsl:if>
      <xsl:if test="$nuuu = 1">
        <xsl:if test="../../service-response/type = 'internal '
            ">
          <misc committed="true"/>
        </xsl:if>
        <xsl:if test="../../service-request/type = 'internal '"
            >
          <misc committed="true"/>
```

```
        </xsl:if>
      </xsl:if>
    </location>
  </xsl:for-each>
</xsl:for-each-group>
<location>
  <portcontainer>
    <nailcontainer>
      <labelcontainer>
        <object>
          <metrics h="40" w="100" x="110" y="140"/>
          <misc id="0"/>
        </object>
        <label text="idle" x="147" y="150"/>
        <font name="SansSerif"/>
      </labelcontainer>
    </nailcontainer>
  </portcontainer>
  <misc initial="true"/>
</location>

<xsl:for-each select="contract/reactivity">
  <xsl:variable name="ren" select="."/>
  <xsl:if test="action">
    <location>
      <xsl:for-each select="service-response">
        <xsl:variable name="n" select="name"/>
        <xsl:for-each select="id">
          <xsl:variable name="renn" select="$ren/id"/>
          <xsl:variable name="re" as="xs:string" select="
              fn:concat(xs:string($renn),"_",xs:string(.))"/>
          <portcontainer>
            <nailcontainer>
              <labelcontainer>
                <object>
```

```
                    <metrics h="40" w="100" x="390" y="70"/>
                    <misc id="{$rc}"/>
                  </object>
                  <label text="{fn:concat($n,$ren/name)}" x="400
                      " y="80"/>
                  <font name="SansSerif"/>
                </labelcontainer>
              </nailcontainer>
            </portcontainer>
            <misc committed="true" tasktype="{fn:concat($n,'T')}
                "/>
          </xsl:for-each>
        </xsl:for-each>
      </location>
    </xsl:if>
  </xsl:for-each>

  <xsl:for-each select="contract">
    <xsl:for-each select="reactivity">
      <xsl:variable name="reactivity-n" select="."/>
      <xsl:for-each select="action">
        <xsl:variable name="action-n" select="."/>
        <xsl:if test="not(exists(to))">
          <location>
            <xsl:for-each select="$reactivity-n/id">
              <xsl:variable name="recNa" select="../name"/>
              <xsl:variable name="reactivity-name" select="."/>
              <xsl:for-each select="$action-n/id">
                <xsl:variable name="result" as="xs:string"
                    select="fn:concat(xs:string($reactivity-name
                    ),_xs:string(.))"/>

                <portcontainer>
                  <nailcontainer>
                    <labelcontainer>
```

```
                    <object>
                      <metrics h="40" w="100" x="20" y="280"/>
                      <misc id="{$result}"/>
                    </object>
                    <label text="{fn:concat($action-n/name,$
                        recNa)}" x="43" y="290"/>
                    <font name="SansSerif"/>
                  </labelcontainer>
                </nailcontainer>
              </portcontainer>
              <misc committed="true" tasktype="{fn:concat($
                  action-n/name,'T')}"/>
            </xsl:for-each>
          </xsl:for-each>
        </location>
      </xsl:if>
    </xsl:for-each>
  </xsl:for-each>
</xsl:for-each>


<xsl:for-each-group select="contract/reactivity/service-
    request" group-by="name">
  <xsl:variable name="nu" select="count(current-group())"/>
  <xsl:variable name="po" select="position()"/>
  <xsl:variable name="i" select="current-group()[1]/id"/>
  <xsl:for-each select="current-group()[1]/name">

    <transition>
      <edge fromport="2" fromvertex="0" toport="1" tovertex="
          {$i}">
        <nailcontainer>
          <labelcontainer>
            <object>
              <metrics w="130" x="210" y="160"/>
```

```
        <misc id="{$po}"/>
      </object>
      <label align="left" x="239" y="123"/>
      <font name="SansSerif"/>
    </labelcontainer>
    <nail cursor="{$po}" id="0" x="207" y="157"/>
    <nail cursor="{$po}" id="1" x="337" y="157"/>
  </nailcontainer>
</edge>
<metrics lx="239" ly="123" sx="210" sy="160"/>

<xsl:variable name="syn">
  <xsl:if test="not(../type = 'internal')">

    <xsl:for-each select="../../service-request/name">
      <xsl:sequence select="fn:concat(xs:string(.),'?')"
          />
    </xsl:for-each>
  </xsl:if>

  <xsl:if test="../type = 'internal'">
    <xsl:if test="$nu = 1">

      <xsl:for-each select="../../service-response/name"
          >
        <xsl:sequence select="fn:concat(xs:string(.)
            ,'!')"/>
      </xsl:for-each>
    </xsl:if>
  </xsl:if>
</xsl:variable>

<xsl:variable name="assig">
  <xsl:if test="../../timeConstraint">
```

```
            <xsl:variable name="vv" select="../../timeConstraint
                /name"/>
            <xsl:sequence select="fn:concat($vv,':=0')"/>
          </xsl:if>
        </xsl:variable>

        <xsl:variable name="GU">

          <xsl:if test="$nu = 1">
            <xsl:if test="../../dataConstraint">
              <xsl:for-each select="../../dataConstraint/
                  constraint">
                <xsl:sequence select="xs:string(.)"/>
              </xsl:for-each>
            </xsl:if>
          </xsl:if>
        </xsl:variable>
        <misc sync="{$syn}" guard="{$GU}" assign="{$assig}"
            showdetails="true"/>
      </transition>
    </xsl:for-each>
  </xsl:for-each-group>

  <xsl:for-each select="contract/reactivity">
    <xsl:variable name="p" select="position()"/>

    <xsl:if test="exists(action)">
      <xsl:variable name="aa" select="id"/>

      <transition>
        <xsl:variable name="from">
          <xsl:for-each select="service-request/id">
            <xsl:sequence select="xs:string(.)"/>
          </xsl:for-each>
        </xsl:variable>
```

```
<xsl:variable name="to">
  <xsl:for-each select="service-response/id">
    <xsl:variable as="xs:string" name="aaaa" select="
        fn:concat(xs:string($aa),xs:string(.))"/>

    <xsl:sequence select="$aaaa"/>
  </xsl:for-each>
</xsl:variable>
<edge fromport="2" fromvertex="{$from}" toport="1"
    tovertex="{$to}">
  <nailcontainer>
    <labelcontainer>
      <object>
        <metrics w="130" x="210" y="160"/>
        <misc id="$p"/>
      </object>
      <label align="left" x="239" y="123"/>
      <font name="SansSerif"/>
    </labelcontainer>
    <nail cursor="$p" id="0" x="207" y="157"/>
    <nail cursor="$p" id="1" x="337" y="157"/>
  </nailcontainer>
</edge>
<metrics lx="239" ly="123" sx="210" sy="160"/>

<xsl:variable name="syn">
  <xsl:if test="not(service-response/type='internal')"
      >
    <xsl:for-each select="service-response/name">
      <xsl:sequence select="fn:concat(xs:string(.),'!')"
          />
    </xsl:for-each>
  </xsl:if>
</xsl:variable>
<xsl:variable name="assig">
```

```
      <xsl:if test="update">
        <xsl:for-each select="update">
          <xsl:variable name="toBeUpdated" select="
              toBeUpdated"/>
          <xsl:variable name="value" select="value"/>
          <xsl:if test="position()=1">
            <xsl:sequence select="fn:concat($toBeUpdated,':
                =',$value)"/>
          </xsl:if>
          <xsl:if test="position()!=1">
            <xsl:sequence select="fn:concat(',',$toBeUpdated
                ,':=',$value)"/>
          </xsl:if>
        </xsl:for-each>
      </xsl:if>
    </xsl:variable>
    <xsl:variable name="GU">
      <xsl:if test="dataConstraint">
        <xsl:for-each select="dataConstraint/constraint">
          <xsl:sequence select="xs:string(.)"/>
        </xsl:for-each>
      </xsl:if>
    </xsl:variable>
    <misc sync="{$syn}" guard="{$GU}" assign="{$assig}"
        showdetails="true"/>
  </transition>
</xsl:if>

<xsl:if test="not(exists(action))">
  <transition>
    <xsl:variable name="from">
      <xsl:for-each select="service-request/id">
        <xsl:sequence select="xs:string(.)"/>
      </xsl;for-each>
    </xsl:variable>
```

```
<edge fromport="2" fromvertex="{$from}" toport="1"
    tovertex="0">
  <nailcontainer>
    <labelcontainer>
      <object>
        <metrics w="130" x="210" y="160"/>
        <misc id="$p"/>
      </object>
      <label align="left" x="239" y="123"/>
      <font name="SansSerif"/>
    </labelcontainer>
    <nail cursor="$p" id="0" x="207" y="157"/>
    <nail cursor="$p" id="1" x="337" y="157"/>
  </nailcontainer>
</edge>
<metrics lx="239" ly="123" sx="210" sy="160"/>

<xsl:variable name="syn">
  <xsl:if test="not(service-response/type = 'internal ')"
     >
    <xsl:if test="not(service-request/type = 'internal ')
       ">
      <xsl:for-each select="service-response/name">
        <xsl:sequence select="fn:concat(xs:string(.)
            ,'!')"/>
      </xsl:for-each>
    </xsl:if>
  </xsl:if>
</xsl:variable>
<xsl:variable name="assig">
  <xsl:if test="update">
    <xsl:for-each select="update">
      <xsl:variable name="toBeUpdated" select="
          toBeUpdated"/>
```

```
      <xsl:variable name="value" select="value"/>
      <xsl:if test="position()=1">
        <xsl:sequence select="fn:concat($toBeUpdated,':
            =',$value)"/>
      </xsl:if>
      <xsl:if test="position()!=1">
        <xsl:sequence select="fn:concat(',',$toBeUpdated
            ,':=',$value)"/>
      </xsl:if>
    </xsl:for-each>
  </xsl:if>
</xsl:variable>

<xsl:variable name="kn" select="service-request/name"/>
<xsl:variable name="nu">
  <xsl:for-each-group select="../reactivity/service-
      request" group-by="name">
    <xsl:for-each select="current-group()[1]/name">
      <xsl:if test="xs:string(.)=$kn">
        <xsl:sequence select="count(current-group())"/>
        </xsl:if>
  </xsl:for-each>
  </xsl:for-each-group>
  </xsl:variable>

<xsl:variable name="GU">
  <xsl:if test="$nu!=1">
    <xsl:if test="dataConstraint">
      <xsl:for-each select="dataConstraint/constraint">
        <xsl:sequence select="xs:string(.)"/>
      </xsl:for-each>
    </xsl:if>
  </xsl:if>
</xsl:variable>
```

```
      <misc sync="{$syn}" guard="{$GU}" assign="{$assig}"
          showdetails="true"/>
    </transition>
  </xsl:if>
</xsl:for-each>

<xsl:for-each select="contract/reactivity">
  <xsl:variable name="react-name" select="id"/>
  <xsl:for-each select="action">
    <xsl:if test="not(exists(to))">
      <transition>
        <xsl:variable name="f">
          <xsl:for-each select="FromId">
            <xsl:variable name="reso" as="xs:string" select="
                fn:concat(xs:string($react-name),_xs:string(.)
                )"/>
            <xsl:sequence select="$reso"/>
          </xsl:for-each>
        </xsl:variable>
        <xsl:variable name="t">
          <xsl:for-each select="id">
            <xsl:variable name="res" as="xs:string" select="
                fn:concat(xs:string($react-name),_xs:string(.)
                )"/>
            <xsl:sequence select="$res"/>
          </xsl:for-each>
        </xsl:variable>
        <edge fromport="2" fromvertex="{$f}" toport="1"
          tovertex="{$t}">
          <nailcontainer>
            <labelcontainer>
              <object>
                <metrics w="130" x="210" y="160"/>
                <misc id="$p"/>
              </object>
```

```
            <label align="left" x="239" y="123"/>
            <font name="SansSerif"/>
          </labelcontainer>
          <nail cursor="$p" id="0" x="207" y="157"/>
          <nail cursor="$p" id="1" x="337" y="157"/>
        </nailcontainer>
      </edge>
      <metrics lx="239" ly="123" sx="210" sy="160"/>

      <xsl:variable name="syn">
        <xsl:if test="not(type = 'internal')">
          <xsl:for-each select="name">
            <xsl:sequence select="fn:concat(xs:string(.)
                ,'!')"/>
          </xsl:for-each>
        </xsl:if>
      </xsl:variable>
      <misc sync="{$syn}" showdetails="true"/>
    </transition>
  </xsl:if>
  <xsl:if test="to">
    <transition>
      <xsl:variable name="f">
        <xsl:for-each select="FromId">
          <xsl:variable name="resss" as="xs:string" select="
              fn:concat(xs:string($react-name), xs:string(.)
              )"/>
          <xsl:sequence select="$resss"/>
        </xsl:for-each>
      </xsl:variable>
      <edge fromport="2" fromvertex="{$f}" toport="1"
          tovertex="0">
        <nailcontainer>
          <labelcontainer>
            <object>
```

```
                    <metrics w="130" x="210" y="160"/>
                    <misc id="$p"/>
                  </object>
                  <label align="left" x="239" y="123"/>
                  <font name="SansSerif"/>
                </labelcontainer>
                <nail cursor="$p" id="0" x="207" y="157"/>
                <nail cursor="$p" id="1" x="337" y="157"/>
              </nailcontainer>
            </edge>
            <metrics lx="239" ly="123" sx="210" sy="160"/>

            <xsl:variable name="syn">
              <xsl:if test="not(type = 'internal')">
                <xsl:for-each select="name">
                  <xsl:sequence select="fn:concat(xs:string(.)
                      ,'!')"/>
                </xsl:for-each>
              </xsl:if>
            </xsl:variable>
            <misc sync="{$syn}" showdetails="true"/>
          </transition>
        </xsl:if>
      </xsl:for-each>
    </xsl:for-each>
  </template>
 </xsl:for-each>
 </times>
</xsl:template>
</xsl:stylesheet>
```