Specification and Verification of Trustworthy Component-Based Real-Time Reactive Systems *

Vasu Alagar and Mubarak Mohammad
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada H3G 1M8
{alagar,ms_moham}@cse.concordia.ca

ABSTRACT

This paper presents a formal methodology for the development of trustworthy real-time reactive systems (RTRS). Safety and security are considered as the two significant properties for trustworthy RTRS. The paper presents an overview of a component-based modeling that involves formal descriptions for trustworthy components. Then, Formal rules are introduced for the automatic generation of behavior protocol based on the formal definitions of trustworthy components. A model-checking method to formally verify security and safety properties in the component model is presented.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specifications—*Methodologies*; D.2.4 [**Software Engineering**]: Software / Program Verification—*Formal methods, Model checking*

General Terms

Design, Security, Verification

Keywords

Trustworthiness, Components, Real-Time Reactive Systems

1. INTRODUCTION

In this paper we explain how trustworthiness can be exploited in the specification and verification of component-based real-time reactive systems (RTRS). In the context of RTRS development we identify *safety* and *security* as the two principal factors contributing to trustworthiness. We propose a verification-oriented design methodology that involves (1) formal specification of component structure and functional/non-functional (trustworthiness) properties, (2) automatic generation of component behavior using the specified structure and restricted by the specified properties, and (3) verification of functional / non-functional component behavior using model checking.

*This research is supported by a Research Grant from Natural Sciences and Engineering Research Council of Canada.(NSERC)

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

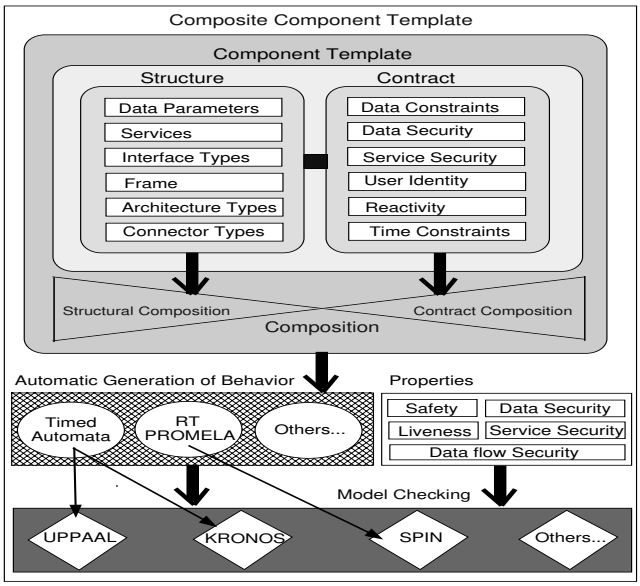

Figure 1: Formal Methodology

Figure 1 depicts our formal methodology. The idea is to formally define the structure of simple components and to define functional and trustworthiness properties at the structural level. There are two benefits for architecting trustworthiness at the structural level. First, it enables the automatic generation of component behavior by analyzing the structure and its properties. Second, it enables reasoning about trustworthiness at the architectural level which is proven to be an important method for attaining trustworthiness [7]. The generated behavior can take different formats depending on defined transformation rules. The defined transformation rules help in (1) automating the process of generating the behavior so that no expertise in behavior specification are required, (2) insuring consistency between the structure and properties defined on one hand and the generated behavior on the other hand, and (3) translating the elements of the structure definitions and the defined properties to a behavior specification format that suits different verification tools. Our goal is to use different verification tools in order to verify a wide range of properties and target different kinds of systems. This is because different verification tools differ in their requirements and abilities [6]: the expressive power of the modeling language, the verification methods used by them, the size and complexity of systems that they can handle, the capabilities suited to different kinds of problems, and the behavior specification format. In this paper we present rules for generating component behavior as ex-

tended timed automata suited for UPPAAL [4] model checker.

In our research, we are focusing on the trustworthiness aspect of RTRS. Reactive systems belong to the class of computer systems that maintain continuous interaction with their environment through stimulus and response. The class of reactive systems in which the response to a stimulus may be strictly regulated by timing constraints is called RTRS. Such systems are required to be trustworthy due to its complexity and the critical contexts it operate in. Although trust is a social concept which is hard to define formally, there is some consensus recently in defining trustworthiness as the degree of user confidence that the system will behave as expected [5]. Safety and security are identified as essential elements for achieving trustworthiness [3]. In the past, research in verifying safety and security properties have progressed in parallel, due to the finding that safety and security can't be formally specified and verified together in any one formal method [8]. We managed to use *component-based development* as a basis for a unified formal model for the specification and verification of safety and security properties of RTRS [3].

Our contributions in this paper are: (1) a formal methodology for developing trustworthy RTRS, (2) transformation rules for the automatic generation of component behavior based on component structure and properties, and (3) model checking safety and security using one method.

2. COMPONENT TEMPLATE - A FORMAL DEFINITION

In our component model, a *component template*, component type, is composed of a *structure* part and a *contract* part. The structure of a template is an abstract external black-box view, called *frame*, and its internal hierarchical structure, called *architecture*. The frame consists of the interface types, the access points to the services provided/requested by the component. Each interface type is associated with a set of services. A service may be parameterized with data parameters. An architecture is a collection of connector types, an abstract view of the tie-ins between interface types. The contract part of the template states the trustworthiness properties required of the system for which the structure is a blue print.

A component is an *instance* of a component template. Every component instantiated from a template has one instance of the structure part defined for the template. The frame of the component is a set of interfaces, where each interface belongs to exactly one interface type in the template frame. It is possible to instantiate multiple interfaces of an interface type. An architecture instance corresponding to a component frame is an instance of the architecture corresponding to the frame in the template, having as many instances of connector types as are required for linking the interfaces in the component. A component's contract constrains the communication pattern at its interfaces and is faithful to the contract part in its template.

In [3] we have introduced a formal component model for trustworthy systems. In this section we present a brief overview of the component model. A component template can be formally specified in terms of its frame and architecture specifications. We focus only on the frame specification because of its relevance to the transformation rules presented in the next section. The internal architecture specification has no impact on the component behavior communicated at the external interfaces.

The frame specification is a tuple $< \Pi, \Sigma, \Lambda, \Xi, \sigma, \Theta, \Gamma, \Omega, \Upsilon, \Psi >$ specifying services, interface types, and properties. The symbol Π denotes a finite non-empty set of interface-types. An *interface* is an instance of an interface type, it inherits the services listed in the type definition. The symbol Σ denotes a finite set of events where each event represents a service provided/requested by the component. The set Σ is divided into a set of input events Σ_{input}, output events Σ_{output}, and internal events $\Sigma_{internal}$ such that $\Sigma = \Sigma_{input} \cup \Sigma_{output} \cup \Sigma_{internal}$ and $\Sigma_{input} \cap \Sigma_{output} \cap \Sigma_{internal} = \emptyset$. An event can carry data parameter values; therefore, we use Λ to denote the finite set of data parameters and define $\Xi : \Sigma \rightarrow \mathbb{P}\Lambda$ as a function that associates with each event a set of data parameters. Events are communicated at the interfaces of the frame; the function $\sigma : \Pi \rightarrow \mathbb{P}\Sigma$ associates a finite non-empty subset of events to each interface-type such that $\forall P, Q \in \Pi, \sigma(P) \cap \sigma(Q) = \emptyset$ i.e. each event is associated with only one interface type. When a request (stimulus) for service is received at an interface, it stimulates the component to perform an action and respond either with an internal processing or with an output event. $\Theta : \Sigma_{input} \rightarrow \Sigma_{output} \cup \Sigma_{internal}$ is a total function that associates a set of possible responses to each request received by the component. The function Θ defines a causality relation between events i.e. $\Theta(e_1) = \{e_2, e_3\}$ means that if event e_1 occurs then event e_2 or e_3 will occur as a response to e_1. The responses of the component can be constrained using (1) time constraints and (2) data parameter constraints. First, Γ denotes the finite set of timing constraints for the events in Σ, where each time constraint involves conjuncts of the form $(t(r) - t(s)) \circ n$, where $t(.)$ is the time function for event occurrences, $s \in \Sigma$ is an input event, $r \in \Sigma$, $r \in \Theta(s)$ is a response to s, $\circ \in \{<, \leq, =, \geq, >\}$, and $n : \mathbb{N}$. Second, Ω denotes a finite set of constraints for the data parameters associated with the events in Σ, where each data constraint of an event $s \in \Sigma$ is a predicate defined over the values of the data parameters $\Xi(s)$ associated with s. If s has n number of responses in $\Theta(s)$ than there must be n number of mutually exclusive data constraints defined over the data parameters of s. This ensures that the responses of s are mutually exclusive. The services provided by the component can be secured and restricted only to authorized users. The introduction of security properties at the frame enriches its behavior by forcing (1) an analysis of the stimulus received before processing it internally, and (2) an analysis of the response before sending it. There are two prerequisites for ensuring security at the interfaces of the component: (1) knowing the identity of the entity on whose behalf the service is requested/provided, henceforth called *user*, and (2) having an explicit definition of an *access control matrix* that defines the *access level* of users to both events and information carried by events. We assume that U denotes the set of users. For the sake of simplicity we assume $AC = \{grant, deny\}$ is the set of access rights for events, and $DA = \{read, write\}$ is the set of allowed actions on data. The function $\Upsilon : U \times \Sigma \rightarrow AC$ defines the event-security access by assigning for every pair $(user, event)$ an authorization which is either $grant$ or $deny$. The function $\Psi : U \times \Lambda \rightarrow \mathbb{P}\ DA$ enforces data-security access. It assigns for every pair $(user, dataparameter)$ an authorization which is a subset of DA. If $\Psi(u, d) = \emptyset$ user u is denied access to data d. The security property is defined in terms of *event-security* and *data-security*. An interface of a component is event-secure if (1) every input event is received from a user who is authorized to trigger the input event, and (2) for every response event sent, the user receiving the response is authorized to view the response. An interface is data-secure if (1) the user has access rights for the data parameters in every stimulus sent by the user, and (2) for every response sent through the interface, the user receiving the response

has access rights for the data parameters in the response.

3. FORMAL VERIFICATION

In this section, we present brief information about UPPAAL model checker. Then, we introduce transformation rules for the automatic generation of component behavior. Finally, we describe how the verification process is conducted using UPPAAL model checker.

3.1 UPPAAL

UPPAAL [4] is a mature model checker that has been used successfully for more than a decade to model check several types of concurrent real time systems. The UPPAAL modeling language is based on timed automata $TA = (L, l_0, K, A, E, I)$ where L is the set of locations denoting states, l_0 is the initial location, K is the set of clocks, A is the set of actions denoting events that cause transitions between locations, E is the set of edges, and I is the set of invariants. Formally, $E \subseteq L \times A \times B(K) \times 2^K \times L$ where $B(K)$ is the set of clock and data constraints denoting guard conditions that restrict transitions, 2^K is the set of clock initializations to set clocks whenever required, and $I : L \rightarrow B(K)$ is a function assigning clock constraints to locations as invariants. UPPAAL extends timed automata with additional features. We present some of those features that are relevant to the this paper:

- **Templates:** Timed automata are defined as templates with optional parameters. Parameters are local variables that are initialized during template instantiation in system declaration.
- **Global variables:** Global variables and user defined functions can be introduced in a global declaration section. Those variables and functions are shared and can be accessed by all templates.
- **Binary synchronization:** Two timed automata can have a synchronized transition on an event when both move to new state at the same time when the event occurs. An event that causes synchronous transition is defined as a *channel*, a UPPAAL data type. A channel can have two directions: input(labeled with ?) and output(labeled with!).
- **Committed Location:** Time is not allowed to pass when the system is in a committed location. If the system state includes a committed location, the next transition must involve an outgoing edge from the committed location.
- **Expressions:** There are three main types of expressions: (1) *Guard* expressions are evaluated to boolean and used to restrict transitions; guard expressions may include clocks and state variables, (2) *Assignment* expressions are used to set values of clocks and variables, and (3) *Invariant* expressions are defined for locations and used to specify conditions that should be always true in a location.
- **Edges:** Edges denote transitions between locations. An edge specification consists of four expressions: *Select:* assigns a value from a given range to a defined variable, *Guard:* an edge is enabled for a location if and only if the guard is evaluated to true, *Synchronization:* specifies the synchronization channel and its direction for an edge, and *Update:* an assignment statements that reset variables and clocks to required values.

In UPPAAL, system properties are expressed formally using a simplified version of CTL [4] as follows:

- **Safety property** is formulated positively stating that some thing good is invariantly true. For example, let φ be a formula, $A\Box\ \varphi$ means that φ should be always true.
- **Liveness property** states that some thing good will eventually happen. For example, $A\diamond\ \varphi$ means that φ will eventually be satisfied.

3.2 Transformation Rules

In this section, we introduce the transformation rules for the automatic generation of component behavior based on the analysis of component's structure and contract defined in the component frame specification. A component-based system is a network of connected components. Every component is mapped to a UPPAAL template in a one to one manner. We assign a parameter to every UPPAAL template to denotes the identifier of the user on whose behalf the component is running. This parameter will be used for ensuring event and data security.

Let $O = \{o_1, \ldots, o_n\}$ be the set of components in a RTRS, $o_i =< \Pi_{I_i}, \Sigma_i, \Lambda_i, \Xi_i, \sigma_i, \Theta_i, \Gamma_i, \Omega_i, \Upsilon_i, \Psi_i >$ such that:$\Sigma_{input} \subseteq \Sigma_i$ denotes the set of stimulus events, $\Sigma_{output} \subseteq \Sigma_i$ denotes the set of output events, $\Sigma_{response} \subseteq \Sigma_{output}$ denotes the set of responses, $\Sigma_{requests} \subseteq \Sigma_i$ denotes the output events sent to other components as requests for services, and $\Sigma_{internal} \subseteq \Sigma_i$ denotes the set of internal events that are local to the component. Let $TA = (L, L_0, K, A, E, I, u)$ be the definition of UPPAAL timed automata where u denotes the user identity parameter associated with the template at its instantiation. Then, the transformation rules construct $T = \{t_1, \ldots, t_n\}$, a set of UPPAAL templates, where t_i is the template constructed from component o_i.

In the definition of a component frame, Π and σ are used in defining the architecture. Therefore, Π and σ don't affect the behavior of the component, hence, are not used in the transformation process. In brief, during the process of constructing $TA = (L, l_0, K, A, E, I)$ from frame specification:

- Σ is used to construct L where every location in L denotes the state of processing an event in Σ,
- Γ is used to construct K and I where a clock in K and an invariant in I are defined for every time constraint in Γ,
- Σ is used to construct A where an action in A is defined for every input or output event in Σ, and
- $\Sigma, \Lambda, \Xi, \Theta, \Omega, \Upsilon$, and Ψ are used to construct E and its associated expressions. More precisely, Λ defines data parameters in Ξ which in turn are used in defining data constraints in Ω that are used along with Υ to define $Guard$ conditions for edges. Σ and Θ are used in defining $Sync$ expression. Ψ is used to control data parameters access in $Update$ expression.

We extend the UPPAAL formal template by adding security features. In the global declaration section, we define: (1) a list of system user identities U, (2) an *event-access control matrix* that defines user access rights to events, (3) a *data-access control matrix* that defines user access rights to events data parameters, (4) an

event security function $EventSecurity: U \times \Sigma \rightarrow boolean$ that searches the event-access control matrix of users-events and returns boolean value indicating whither the user has access or not, (5) a data security function $DataSecurity: U \times \Lambda \rightarrow boolean$ that searches the data-access control matrix of users-data and returns a boolean indicating whither the user has the proper access right (*write* for stimulus parameters and *read* for response parameters) or not.

An informal discussion of the steps for constructing $TA = (L, L_0, K, A, E, I, u)$ is given bellow:

Locations [L]. : A component provides and requests a set of services. The details of service processing are hidden behind component interfaces. Therefore, we use locations to denote the states for processing services. Services are abstracted as events. The function $\Delta: \Sigma \rightarrow L$ constructs for each event a location $\Delta(e)$ in L. The location is the state for processing the event e. The set of locations L can be constructed with the help of Σ as follows:

- [L.1] Create an initial location l_0 to denote the *idle* state where the component is waiting for a stimulus.
- [L.2] Stimulus events correspond to the services provided by the component. For every stimulus event, create a location to represent the service of processing the stimulus.
- [L.3] Output events that are not responses to stimulus correspond to the services requested by the component. For every output event that is not a response to a stimulus create a location.

Clocks [K]. : Time constraints in Γ can be represented by clocks in K and invariants representing clock constraints in I. The set of clocks K can be constructed by creating a clock for every time constraint that constrains the response of a stimulus. Clocks are defined as template's local variables.

Invariants [I]. : Time constraints are defined as location invariants in I. We create an invariant in I for each time constraint in Γ and assign it to $\Delta(e)$.

Actions [A]. : The set of actions A can be constructed by creating an action in A for every input and output event in Σ. Actions are defined as synchronous channels. Input actions are decorated with ? and output actions are decorated with !.

Edges [E]. : The behavior of a component is based on stimuli and responses. Therefore, E can be constructed using Σ according to the rules [E.1], [E.2], and [E.3] defined bellow. The specification of edge expressions is derived from the data parameters Ξ and the constraints Ω, Υ, and Ψ that are related to the action a, which causes the transition, according to the following rules [E.Ex]:

- *Select*: It is used to get a value in a temporary variable for each event data parameter in $\Xi(a)$. These values will be assigned to their corresponding data parameters in the *Update* expression.
- *Guard*: A guard condition is a conjunction $Pr_1 \wedge Pr_2$ such that $Pr_1 \in \Omega$ which is a predicate on data parameters in $\Xi(a)$ and $Pr_2 \in \Upsilon$ which is the event security related to a.
- *Sync*: the action, the event causing the transition.
- *Update*: It includes assignment statements that update data parameters in $\Xi(a)$ and reset the clock in K related to the time constraint in Γ that is defined for a. In order to ensure data security, update statements are constrained by $DataSecurity$ function as follows:
  $\forall d \in \Xi(a),\ d := DataSecurity(u, d)?Select(d) : Null$,
  which means that if the user u has access to the data parameter d then d will be assigned the selected value; otherwise, d will be set to $Null$.

The following rules are used to construct template edges. After constructing each edge, the rules in [E.Ex] are used to define its expressions.

- [E.1] For every stimulus e create an edge from the initial location l_0 to $\Delta(e)$. If $\Theta(e)$ is time constrained then we should reset the clock.
  After finishing the processing of e by sending $\Theta(e)$, the component can go back to idle state waiting for the next stimulus. Therefore, for every response, we create an edge from $\Delta(e)$ back to l_0.
- [E.2] In order to provide the required services, the component may request services from other components. When a stimulus e has a response $\Theta(e) \in \Sigma_{request}$ then create an edge from $\Delta(e)$ to $\Delta(\Theta(e))$ and a second edge from $\Delta(\Theta(e))$ to l_0 .
- [E.3] the component may have a concurrent behavior. It can receive stimuli while processing others. Therefore, we create an edge from every location that represents stimulus processing location l_{p1} to the other stimulus processing locations l_{p2}. Use intermediate committed locations and split the edge into two edges: (1) an edge from l_{p1} to the committed location labeled with the stimulus and (2) an edge from the committed location to l_{p2} labeled with the response of l_{p1}. The reason for having two edges is that UPPAAL doesn't allow having two synchronous channels on an edge.

EXAMPLE 1. *Let* $< \Pi, \Sigma, \Lambda, \Xi, \sigma, \Theta, \Gamma, \Omega, \Upsilon, \Psi >$ *be a frame specification where* $P = \{p_1\}$*;* $\Sigma = \{e_1, e_2, e_3\}$ *such that* $\Sigma_{input} = \{e_1\}$, $\Sigma_{response} = \{e_2\}$, $\Sigma_{request} = \{e_3\}$*;* $\Lambda = \{d\}$*;*$\Xi(e_1) = \{d\}$*;* $\sigma(p_1) = \{e_1, e_2, e_3\}$*;* $\Theta(e_1) = e_2$, $\Theta(e_1) = e_3$*;* $\Omega(e_1, e_2) : d > 10$, $\Omega(e_1, e_3) : d \leq 10$*;* $\Gamma(e_1, \theta(e_1)) = [0, 5]$*;* $U = \{u_1\}$, $\Upsilon(u_1, e_1) = \Upsilon(u_1, e_2) = \Upsilon(u_1, e_3) = grant$, $\Psi(u_1, d) = \{read, write\}$. *Figure 2 shows the extended time automata generated for this example using the transformation rules. The construction is done as follows:*
Locations*:* l_0 *is created according to rule [L.1],* l_1 *according to [L.2], the invariant at* l_1 *according to [I], and* l_2 *according to [L.3].*
Edges*: created according to the following rules and [E.Ex]: (1)* (l_0, e_1, l_1) *is created according to [E.1], (2)* (l_1, e_2, l_0) *is created according to [E.1], (3)* (l_1, e_3, l_3) *is created according to [E.2], and (4)* (l_2, e_3, l_0) *is constructed according to [E.2].*

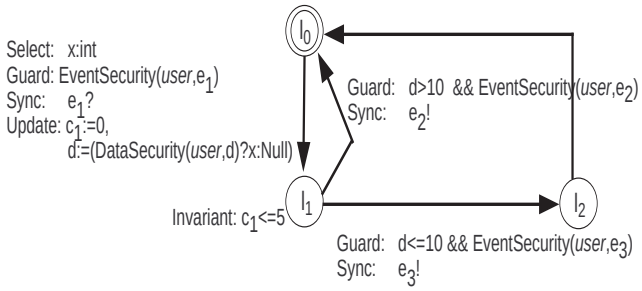

Figure 2: Example

Clocks*: c_1 and the invariant at l_1 are created according to rules [K] and [I].*
Actions*: $e_1?, e_2!, e_3!$ are created according to [A] and [E.Ex]*

3.3 Verification Process

In [2] we have applied the methodology successfully to specify and model check a simplified version of the steam boiler controller case study [1]. The system consists of 3 components: (1) *controller* has 10 locations, (2) *level measuring* has 3 locations, and (3) *monitoring* has 5 locations. The steps of performing the verification process are:

- using UPPAAL *editor*, we specified the components as UPPAAL templates using the automatic transformation rules. Then, in the system declaration section of the editor, we created instances of the templates and defined the RTRS as the parallel composition of the instances,
- using UPPAAL *verifier*, we specified safety, liveness, event security, and data security properties.
  - **Event security**: An event can be triggered only by a user whose access level is *grant*. This is expressed as: $A\Box$ *for all(i:int[1,NoOfUsers]) C.user==i &&* $C.event_x$ *imply EventSecurity(i,*$event_x$*)==grant*. It means: invariantly, in all system executions, $event_x$ can be triggered by authorized users only.
  - **Data security**: A data parameter value should be visible only to authorized users. This is expressed as the invariant: $A\Box$ *for all(i:int[1,NoOfUsers]) C.user==i && DataParameter!=Null imply DataSecurity(i,DataParameter) ==read*. It means: invariantly, in all system executions, the value of $DataParameter$ can be visible only to authorized users; otherwise, it is set to $Null$.
- We executed the model checker to verify the properties against the defined system.

The experiment was performed on two machines: (1) An average PC workstation with 512MB of memory and Pentium IV processor running *Windows XP Home Edition*, and (2) a powerful server with 3GB of memory and Pentium Xeon 3GH running *Windows Server 2003*. Table 1 presents the time duration of model checking each property using the two machines. The time ranges between 1 to 2 minutes on the workstation.

Table 1: Time Duration of Model Checking

Result	workstation	server
Safety	1.49 min	0.12 min
Liveness	1.29 min	0.12 min
Event Security	1.21 min	0.11 min
Data Security	2.06 min	0.12 min

4. CONCLUSION

We have introduced (1) a formal methodology for developing trustworthy systems and (2) formal set of rules for generating the behavior of a component-based model, and (3) model check functional and non-functional properties using UPPAAL model checker. We have applied our method for a simple version of the steam boiler controller problem. We plan to evaluate our method on problems from different domains where both safety and security are critical. Our research directions include: (1) investigating the requirements of an ADL for expressing trustworthiness and (2) building a visual interface tool that enables software architects to specify trustworthy component-based systems. Then, we will derive the formal description automatically from the visual notations and generate system behavior in different formats. The generated behavior will be input into model checkers to perform the verification process. This will hide the complexity of formal specification and enable software architects to easily design and verify trustworthy systems.

5. REFERENCES

[1] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack. Formal methods for industrial applications, specifying and programming the steam boiler control. London, UK, 1996. Springer-Verlag.

[2] Vasu Alagar and Mubarak Mohammad. A formal approach for the development of trustworthy component-based rtrs - case study. http://users.encs.concordia.ca/[tilda]ms_moham/sv.pdf.

[3] Vasu Alagar and Mubarak Mohammad. A component model for trustworthy real-time reactive systems development. In *FACS'07*, Sophia-Antipolis, France, Sept 2007.

[4] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In *Proceedings of SFM-RT'04*, 2004.

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

[6] John C. Knight Elisabeth A. Strunk, M. Anthony Aiello. A survey of tools for model checking and model-based development. Technical Report CS-2006-17, Dept. of Computer Science, University of Virginia, June 2006.

[7] Cristina Gacek and Rogrio de Lemos. *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, chapter Architectural description of dependable software systems, pages 127–142. Springer London, 2006.

[8] John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Trans. on Software Engineering*, 22(1):53–67, 1996.