Testing time goal-driven requirements with Model Checking Techniques

Gregorio Díaz, Elena Navarro, María-Emilia Cambronero, Valentín Valero and Fernando Cuartero
University of Castilla-La Mancha - Department of Computer Science
Address: Escuela Politécnica Superior de Albacete. 02071 - SPAIN
Email: [gregorio,enavarro,emicp,valentin,fernando]@dsi.uclm.es

Abstract

In this paper we present the testing of time goal requirements by using Model Checking as an engine for test generation. The goal model that we use to capture the time requirements is an extension of KAOS. This goal model specifies the properties that the system must satisfy and how they are tested by using the model checker engine, UPPAAL. As an illustration of this proposal we take a particular study case based on GSM cell technology.¹

1 Introduction

In software and hardware development the engineers provide a concrete model of the system behavior. This model is naturally restricted with correctness and performance constraints that have to be checked by the verification software. Another alternative for verifying system is software testing. Software and hardware testing is the process that tests the functionality and correctness of the system through execution or analysis. As noted by Callahan et al in [6] software testing could be used properly in conjunction with model checking to develop high quality software and hardware systems.

Although software and hardware testing can guarantee high quality of software and hardware development, they cannot provide a “correct” system in isolation. On the other hand, the introduction of Formal approaches such as Model Checking, makes this objective feasible. Thus, it becomes important for testing frameworks to incorporate these formal approaches. This has meant a there is growing consensus on the use of formal methods, i.e. development methods based on formalism, which could have significant benefits in testing systems due to the enhanced rigor these methods bring [12]. Furthermore, these formalisms allow us to reason with the constructed models, analyzing and verifying some properties of interest of the described systems. One of these formalisms is timed automata [3], which are very frequently used in model checking [1], and there are some well-known tools supporting them, like UPPAAL [11, 8, 14] or KHRONOS [16] either SPIN [13]. The Model Checking performs the verification of models that are derived form the system behaviors. The specification and definition of properties to be verified are an important key. In this work, we use an extension of KAOS [2, 7, 19], an approach for goal-oriented for requirement engineering, in order to formally specify the system requirements. Our approach uses a combination of Model Checking in conjunction with goal-oriented for property gathering. This approach is depicted in figure 1. The goal oriented model and the system model are extracted from the real system and checked by UPPAAL. The checking process may find errors that are model traces that are visualized as counterexamples. These counterexamples are model traces that show paths that the model follows to reach the error. Finally these errors are tested in the real system. Thus this diagram depicts a methodology for test generation by using Model Checking as a tester generation engine.

¹This work has been supported by the CICYT project “Supported by the spanish government (cofinanced by FEDER founds) with the project “Application of Formal Methods to Web Services”, with reference TIN2006-15578-C02-02, and the JCCLM regional project “Application of Formal Methods to the Design and Analysis of Web Services and e-commerce applications”, with reference PAC06-0008-6995.”
to a GSM-based communication system, and the UPPAAL tool is used to test and analyze the goal established in the previous one. Finally, the conclusions and the future work are presented in Section 5.

2 Uppaal as a tester engine

UPPAAL is a suite tool that performs symbolic model checking over timed automata. The time properties that it is possible to specify by UPPAAL are based on the temporal logic subset of timed computation tree logic (TCTL). Now, we will see the timed automata, the properties and the test algorithm in detail.

2.1 Timed Automata

By definition, a timed automaton is a standard finite-state automaton extended with a finite collection of real valued clocks. The clocks are assumed to proceed at the same rate and their values may be compared with natural numbers or reset to 0. UPPAAL extends the notion of timed automata to include integer variables, i.e. integer valued variables that may appear freely in general arithmetic expression used in guards as well as in assignment. Note that for backward reachability analysis, which we will define after, the variable assignment is restricted to any value of the form $ax + b$ where $a, b \in \mathbb{Z}$ and $x$ is the variable being reassigned, whereas for forward reachability analysis there is no such restriction.

The model also allows clocks not only to be reset, but also to be set to any non-negative integer value.

Definition 1 (Atomic Constraints) Let $C$ be a set of real valued clocks an $I$ a set of integer valued variables. An atomic clock constraint over $C$ is a constraint of the form: $x \sim n$ or $x - y \sim n$, for $x, y \in C$, $\sim \in \{=, \geq, \leq, \} \text{and } n \in \mathbb{N}$. An atomic constraint over $I$ is a constraint of the form: $i \sim n$, for $i \in I$, $\sim \in \{=, \geq, \leq\} \text{and } n \in \mathbb{Z}$.

By $C_c(C)$ we will denote the set of all clock constraints over $C$, and by $C_i(I)$ we will denote the set of all integer constraints over $I$.

Definition 2 ( Guards) Let $C$ be a set of real valued clocks, and $I$ a set of integer valued variables. A guard $g$ over $C$ and $I$ is a formula generated by the following syntax: $g := c l g \land g$, where $c \in (C_c(C) \cup C_i(I))$.

$B(C, I)$ will stand for the set of all guards over $C$ and $I$.

Definition 3 ( Assignments) Let $C$ be a set of real valued clocks and $I$ a set of integer valued variables. A clock assignment over $C$ is a tuple $(v, c)$, where $v \in C$ and $c \in \mathbb{N}$. An integer assignment over $I$ is a tuple $(w, d)$ representing the assignment $w = d$, where $w \in I$ and $d \in \mathbb{Z}$.

We will use $A(C, I)$ to denote the power-set of all assignments over $I$ and $C$.

Definition 4 (Timed automata) A timed automaton $A$ over a finite set of actions $Act$, clocks $C$ and integer variables $I$ is a tuple $\langle L, l_0, E, V \rangle$, where $L$ is a finite set of nodes (control-nodes), $l_0$ is the initial node, $E \subseteq L \times B(C, I) \times Act \times A(C, I) \times L$ corresponds to the set of edges, and $V : L \rightarrow B(C, I)$ assigns invariants to locations. For a brief notation, we will denote $l \xrightarrow{g.a.r} l'$ by the edge $\langle l, g, a, r, l' \rangle \in E$.

Definition 5 (Synchronization Function) Let $T \subseteq Act \times Act$ be a function such that:

$$\langle a_1!, a_2 ! \rangle \in T \Rightarrow \langle a_1 ?, a_1 ! \rangle \in T \text{ for all } a_i$$

Definition 6 (Parallel Composition) Let $A_1, A_2$ be two timed automata. The parallel composition $(A_1 \mid A_2)$ is a timed automaton $\langle L, l_0, E \rangle$, where $(l_1, l_2) \in L$ whenever $l_1 \in L_1$ and $l_2 \in L_2$, $l_0 = \langle l_1, 0 \rangle$, and the edges $E$ are defined as follows:

- $\langle (l_1, l_2) \xrightarrow{g.a.r} (l_1', l_2) \rangle$ if $\langle (l_1, l_2) \xrightarrow{g_{a1}.r_{1}} (l_1', l_2') \rangle$ and $\langle (l_2, l_2) \xrightarrow{g_{a2}.r_{2}} (l_2', l_2) \rangle$.
- $\langle (l_1, l_2) \xrightarrow{g.a.r} (l_1', l_2') \rangle$ if $\langle (l_1, l_2) \xrightarrow{g.a.r} (l_1, l_2') \rangle$ and $\langle (l_2, l_2') \xrightarrow{g.a.r} (l_2, l_2) \rangle$.

A state of a timed automaton $A$ is a pair $\langle l, u \rangle \in V(l)$, where $l$ is a node of $A$ and $u$ is an assignment, mapping each clock in $C$ to a value in $\mathbb{R}_+$, and each integer variable in $I$ to a value in $\mathbb{Z}$, and $u \in V(l)$ means that $u$ satisfies the invariant of the node $l$. We will use $g(u)$ to denote that the assignment $u$ satisfies the guard $g$. The initial state of $A$ is $\langle l_0, u_0 \rangle$, where $u_0$ is the assignment mapping all variables and clocks to 0.

The evolution of a timed automaton, from a state to another state can proceed by two types of transitions:

- Delay transition: $\langle l, u \rangle \xrightarrow{e(d)} \langle l, u' \rangle$.

- Action transition: $\langle l, u \rangle \xrightarrow{g.a.r} \langle l', u' \rangle$.

Definition 7 (Delay transitions) Let $\langle l, u \rangle$ and $\langle l', u' \rangle$ be two states of a timed automaton $A$, and let $d$ be a positive real. Then

$$\langle l, u \rangle \xrightarrow{e(d)} \langle l', u' \rangle \text{ iff } u' \in \begin{cases} l' = l & \\ V(l') \land \begin{cases} u'(x) = u(x) + d & \text{if } x \in C \\ u'(x) = u(x) & \text{if } x \in I \end{cases} & \end{cases}$$
\textbf{Definition 8} (Action transition) Let \(\langle l, u \rangle\) and \(\langle l', u' \rangle\) be two states of a timed automaton \(A\). Then
\[
\langle l, u \rangle \xrightarrow{\text{guard}, \text{action}} \langle l', u' \rangle \text{ iff } u' \in V(l') \land g(u) \land \begin{cases} 
c_o & \text{if } x \in C \land (x, c_0) \in r 
c_1 & \text{if } x \in I \land (x, c_1) \in r 
u(x) & \text{otherwise}
\end{cases}
\]

\section{2.2 Specifying time properties}

The language that UPPAAL uses to perform the verification is a subset of timed computation tree logic (TCTL) [18, 17], where atomic expressions are location names, variables and clocks from the modeled system. The properties are defined using local properties that are either true or false depending on a specific configuration.

\textbf{Definition 9} (Local Property) Given an UPPAAL model \(\langle \vec{A}, \text{Vars}, \text{Clocks}, \text{Chan}, \text{Type} \rangle\). A formula \(\varphi\) is a local property iff it is formed according to the following syntactical rules:

\[
\varphi ::= \text{deadlock} \mid \text{A} . l \mid x \triangleright c \mid x \triangleright y \triangleright c \mid a \triangleright b \mid (\varphi_1) \text{ not } \varphi \mid \varphi_1 \text{ or } \varphi_2 \mid \varphi_1 \text{ and } \varphi_2 \mid \varphi_1 \text{ imply } \varphi_2
\]

In Definition 9 we have expressed the syntax of the temporal logic that UPPAAL uses. Now, let us see the definition of the five different property classes that UPPAAL may test.

\textbf{Definition 10} (Temporal Properties) let \(M = \langle \vec{A}, \text{Vars}, \text{Clocks}, \text{Chan}, \text{Type} \rangle\) be an UPPAAL model and let \(\varphi\) and \(\psi\) be local properties. The correctness of temporal properties is defined for the classes \(A[ ]\), \(A <>\) and \(--->\) as follows:

\[
M \models A[ ] \varphi \iff \forall \{\vec{l}, e, v\}^K \in \tau(M).
\]

\[
M \models A <> \varphi \iff \forall \{\vec{l}, e, v\}^K \in \tau(M).
\]

\[
M \models \varphi --> \psi \iff \forall \{\vec{l}, e, v\}^K \in \tau(M).
\]

\[
M \models \varphi \land \psi \iff \forall \{\vec{l}, e, v\}^K \in \tau(M).
\]

The two temporal property classes dual to \(A[ ]\) and \(A <>\) are defined as follows:

\[
M \models E <> \varphi \iff \neg(M \models \neg \text{loc } A[ ] \text{ not}(\varphi))
\]

\[
M \models E [ ] \varphi \iff \neg(M \models \neg \text{loc } A <> \text{ not}(\varphi))
\]

\section{2.3 The tester algorithm}

The tester algorithm that UPPAAL uses is based on the symbolic model checking [4, 15] that uses constraint solving. The algorithm checks if a state in a timed automata is reachable from the initial state or not.

When searching the state space we need two buffers that we can call “wait” and “passed” respectively. The wait buffer holds the states not yet explored and the passed buffer holds the states explored so far.

\textbf{Algorithm 1} Forward Reachability Analysis

If we do forward reachability analysis we initially store \(\langle \vec{l}_0, \vec{u}_0 \rangle\) in the wait buffer. We then repeat the following:

1. Pick a state \(\langle \vec{l}_i, \vec{u}_i \rangle\) from the wait buffer.
2. Check if \(\vec{l}_i = \vec{l}_f \cup \vec{u}_i \subseteq \vec{l}_f\). If that is the case, return the answer yes.
3. If \(\vec{l}_i = \vec{l}_j \cup \vec{u}_i \subseteq \vec{u}_j\), for some \(\langle \vec{l}_i, \vec{u}_i \rangle\) in the passed buffer drop \(\langle \vec{l}_i, \vec{u}_i \rangle\) and go to step 1. Otherwise save \(\langle \vec{l}_i, \vec{u}_i \rangle\) in the passed buffer. If \(\vec{u}_j \subset \vec{u}_i\) we can replace the state \(\langle \vec{l}_i, \vec{u}_j \rangle\) with \(\langle \vec{l}_i, \vec{u}_i \rangle\). (To save space)
4. Find all \(\vec{l}_k\) that are reachable from \(\vec{l}_i\) in one step regardless of the assignments, taking only actions into account. Let \(g_k\) be the set of guards on the performed transition an \(a_k\) the set of resets
5. Now set \(U_k = \text{reset}(sp(U_i) \cap g_k, a_k)\). If \(U_k \neq \emptyset\), store \(\langle \vec{l}_i, \vec{u}_k \rangle\) in the wait buffer.
6. If the wait buffer is not empty go to step 1, otherwise return the answer no.

\section{3 The Goal Model for time requirements}

The requirements, properties and characteristic of the system must be gathered in order to be tested. However, they must be expressed in a formalized manner. There are several languages, graphical diagrams, etc. to perform it,
but we apply those in which the time requirements are well captured. In this sense, goal oriented requirements engineering emerges as a natural choice.

The key activity in goal-oriented requirements engineering is the construction of the goal model. Goals are objectives the system under construction must achieve. Goal formulations thus refer to intended properties to be ensured. They are formulated at different levels of abstraction from high-level, strategic concerns to low-level technical concerns.

Goal models also allow analysts to capture and explore alternative refinements for a given goal. The resulting structure of the goal model is an AND-OR graph. The specific goal-oriented framework considered here is an extension of KAOS methodology \[2, 5, 7, 19\] which has a two-level language: (1) an outer semi-formal layer for capturing, structuring and presenting requirements engineering concepts; (2) an inner formal assertion layer for their precise definition and for reasoning about them.

Two key elements are used as building elements for the definition of a goal model: goals and requirements. A goal prescribes intended behaviors of the system. It may refer to services to be provided (functional goals) or to the quality of service (non-functional goals). A requirement is a leaf goal that requires the cooperation between different parties, which are called agents. Agents are active components that play a role in achieving goal satisfaction.

To Build Goal Models, goals are organized in an AND/OR refinement - abstraction hierarchy where higher-level goals are, in general, strategic, coarse-grained and involve multiple agents whereas lower-level goals are, in general, technical, fine-grained and involve fewer agents. In such structures, AND-refinement links relate a goal to a set of subgoals (called refinement) possibly conjoined with domain properties; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal, as seen in Figure 2. OR-refinement links may relate a goal to a set of alternative refinements, as seen in Figure 3.

Requirements must be checked by the model checker. During the check process, if the system model does not satisfy a requirement then this requirement must be tested over the real system using the counterexample (see figure 1). Requirements are formalized in a real-time temporal logic that we have shown above. Keywords such as Achieve (reachability), Avoid (not safety), Maintain (safety), possibly always, inevitably and unbounded response, are used to name goals according to the temporal behavior pattern they prescribe. They are depicted in the goal model as follows:

![Temporal Behavior and Goal Model Representation](image)

Figure 4 shows a goal model fragment of an aircraft control system. The goal Maintain (safety) \[SafeDoors\] is refined by two OR-refinement links that inherit the safety behavior. The first leaf goal Maintain[DoorsLockedWhileMoving], may be annotated with the following temporal logic assertion stating that in every future state the plane doors shall be locked when the plane is moving. The \[SafeDoors\] formulation will be:

\[ A[] (Plane.Moving \land Doors.Locked). \]

The second leaf goal: Maintain Inevitably [DoorUnlockedWhenEmergency], may be annotated with the following temporal logic assertion stating that in every future state the plane doors shall be unlocked when emergency occurs:

The alternative refinement for the safety subgoal [NoPlaneCollisionOnTakeOff&Landing] which leads to totally different designs when refined: block-based design for NoPlaneInSameRunway and speed control design for WorstCaseFlyToFlyDelayMaintained, whose definitions are, respectively:

\[ A[] \neg (Runway.Occupied) \] and
\[ A[] \text{Runway.FlyToFlyClock < FlyToFlyDelay} \]

The leaf goal [NoDelays] specifies the intended behavior of "Possibly Always, the aircraft control does not suffer delays" which is defined by the formula:

\[ E[] \neg (AircraftControlSystem.Delay) \]

The temporal behavior for plane progression is specified by the Achieve (reachability) subgoal [PlaneProgress] which is refined by two subgoals that inherit the achieve behavior: [Land&TakeOffWhenSignal] and [SignalSetGo] defined by

\[ E <> (Plane.LandorPlane.TakeOff) \land Signal.Go \]
and
\[ E <> Plane.Ready \implies Signal.Go \]

Goals refer to objects or entities which may be incrementally designed from goal specifications to produce a structural model of the system that is specified as an automata. Objects have states defined by the values of their attributes and associations to other objects. In the above formalization of the goal DoorsLockedWhileMoving, Moving and doorsState are attributes of the Train and doors entities that will be declared in the system design.

Figure 5. The GSM architecture

4 Case Study

Our example is based on the Global System for Mobile communications (GSM) and centred on two components of the GSM cell architecture, Mobile Stations and Base Station Subsystem. A GSM is composed of several functional entities, whose functions and interfaces are specified in Figure 5.

Mobile Station, MS for short, consists of the mobile equipment (the terminal) and a smart card called the Subscriber Identity Module (SIM). The SIM provides personal mobility, so that the user can have access to subscribed services irrespective of a specific terminal. By inserting the
SIM card into another GSM terminal, the user is able to receive calls at that terminal, make calls from that terminal, and receive other subscribed services.

The mobile equipment is uniquely identified by the International Mobile Equipment Identity (IMEI). The SIM card contains the International Mobile Subscriber Identity (IMSI) used to identify the subscriber to the system, a secret key for authentication, and other information. The IMEI and the IMSI are independent, thereby allowing personal mobility. The SIM card may be protected against unauthorized use by a password or personal identity number.

**The Base Station Subsystem**, BSS for short, is composed of two parts, the Base Transceiver Station (BTS) and the Base Station Controller (BSC). These communicate across the standardized Abis interface, allowing (as in the rest of the system) the operation between components made by different suppliers.

The Base Transceiver Station houses the radio transceivers that define a cell and handles the radio-link protocols with the Mobile Station. In a large urban area, there will potentially be a large number of BTSs deployed; thus the requirements for a BTS are ruggedness, reliability, portability and minimum cost.

The Base Station Controller manages the radio resources for one or more BTSs. This includes radio-channel setup, frequency hopping and handovers, as described below. The BSC is the connection between the mobile station and the Mobile service Switching Center (MSC).

### 4.1 Timed Automata Specification

This automata specification deals with the relationship between MS and BTS components. It captures the behavior of a data transmission from a BTS component to a MS component. This transmission process often has time restrictions that determine the time interval within which the communication is available. Figures 6 and 7 depict the automata that captures the MS and BTS behaviors, respectively. Figure 6 captures the MS behavior that consists of three states interconnected by transitions that define the communication protocol with a BTS partner. Figure 7 is the specification automaton of a BTS component that consists of two states interconnected also by several transitions.

Basically, the example shows a MS component that is waiting to receive an attach message from a BTS component. Once the MS has received this message, it is ready to receive a data transmission from this BTS component. If no data from the BTS is received within 1000 milliseconds then the MS changes its state to StandBy. Furthermore, if no data is incoming in another interval of 1000 milliseconds then the MS changes to idle and the process starts again. The BTS automaton may transmit by using the transmit message or finishes the transmission by using the detach message and then the BTS and MS automata changes to idle at the same time.

An important tool for capturing the temporal behavior is the clocks, and in this example, we have used two clocks, x and y. These clocks are used in guards, invariants and assignments. Guards are placed on transitions and determines when a transition is enabled. For instance, we can observe in our example the guard in the transition that connects the “StandBy” and “Idle” states of the MS automaton in figure 6. This guard enables the transition when the x clock reaches 2000 time units. Invariants are placed on states and specify the time that it is possible to stay in this state. Assignments are placed on transitions and modify either a variable or a clock when the transition is made. We also can observe an assignment in the transition between the “Idle” and “Transmitting” states in the BSS automaton in figure 7. This assignment resets the clock x when the transition is carried out.

The channels are other components of our automata and
also a key component to specify communications. For instance, in this case we have used three channels attach: transmit and detach. The symbols “!” and “?” which are placed after channels, determine that the channel is being used to synchronize two automata. The synchronization party that uses the “!” symbol is the one that initialize the communication and the one marked with “?” symbol receives the communication. Note that the communication must be established by using the same channel and no other. Thus the communication is performed when a pair of “!” and “?” symbols appear over the same channel.

4.2 Testing time requirements

Figure 8 depicts the goal model for the time requirements that our example must satisfy. The root goal [CorrectCellSystem4GSM] is divided in two subgoals by using an AND-refinement. The first subgoal is [PropperBTSBehavior] and the second one is [PropperMSBehavior].

The [PropperBTSBehavior] subgoal has the temporal character of a maintain goal, which implies that it must always be satisfied. This subgoal consists of an OR-refinement with an other two leaf goals [BTSTransmitsWithinTheInterval] and [BTSNoTransmitsOutsideTheInterval], which inherit the maintain character from [PropperBTSBehavior]. Note that the OR-refinement stipulate that it is enough if only one of these leaf goals is satisfied in order to satisfy the super goal [PropperBTSBehavior].

The [BTSTransmitsWithinTheInterval] leaf goal specifies that the temporal property of the BTS component always transmits data within the interval of 2000 milliseconds and is specified as follows:

\[ \forall \Box (BTS.Transmit \Rightarrow BTS.y <= 2000) \] (1)

The [BTSIdleOutsideTheInterval] leaf goal specifies that the temporal property of the BTS component is idle (or does not transmits data) out the interval of 2000 milliseconds and is specified as follows:

\[ \forall \Box (BTS.y > 2000 \Rightarrow BTS.Idle) \] (2)

The [PropperMSBehavior] subgoal has the temporal character of an achieve goal and consists of an AND-refinement in two subgoals [PropperMSReception] and [PropperMSSStops] that inherit the achieve character. The [PropperMSReception] subgoal is divided by an AND-refinement in two leaf goals with an unbounded response character that are labeled as [MSWillBeReadyOnTime] and [MSStandByForTheInterval]. The [PropperMSSStops] consists of an AND-refinement with two unbounded response characterized leaf goals named [MSStopsWhenDetach] and [MSStopsWhenNoData].

The [MSWillBeReadyOnTime] leaf goal states that the property of the MS will be ready on time to receive data from a BTS component and is specified as follows:

\[ MS.Idle \rightarrow ((MS.Ready \land MS.x <= 1000) \lor MS.Idle) \] (3)

The [MSStandByForTheInterval] leaf goal states that the property of the MS will stay in the Stand By state during 1000 milliseconds if no data is transmitted and is specified as follows:

\[ MS.Ready \rightarrow ((MS.StandBy \land MS.x >= 1000 \land MS.x <= 2000) \lor MS.Idle \lor MS.Ready) \] (4)

The [MSStopsWhenDetach] leaf goal states that the property of the MS will stop when receives the detach message and is specified as follows:

\[ MS.Ready \rightarrow ((MS.Idle \land MS.x > 1000) \lor MS.Ready \lor MS.StandBy) \] (5)

The [MSStopsWhenNoData] leaf goal defines the property of the MS will be ready on time to receive data from a BTS component and is specified as follows:

\[ MS.StandBy \rightarrow ((MS.Idle \land MS.x >= 2000) \lor MS.StandBy) \] (6)

After the specification of these six leaf goals, they must be tested in the timed automata that we have seen in Figures.

![Figure 9. The counterexample trace generated by UPPAAL](image-url)
6 and 7. To test the automata we use the model checker UPPAAL and the output is “The five first properties 1, 2, 3, 4 and 5 that represent some leaf goals of the goal model depicted in figure 8 are satisfied by the automata. On the other hand, the last one, 6, is not satisfied”. By analyzing this last property the counterexample that UPPAAL provides shows that the error is found when the Mobile Station component waits in the state Stand By for more than 2000 milliseconds and then a deadlock occurs in the model. As an illustration of this analysis Figure 9 shows the trace of this counterexample generated by UPPAAL.

This conclusion shows us which there are errors in our system model. Thus, this last property 6 is a good test must be checked in the real GSM scenario.

5 Conclusions and Future Work

The use of model checking in the test generation provides the testing of software and hardware with a powerful tool that minimizes its cost. One key point, in this sense, is the automatic generation of counterexamples helpful for testing real systems. In addition, it allows us to minimize the temporal cost of an extensive testing so that the time to make a complete test over the real system is greater than over the system model.

Furthermore, the use of goal-oriented requirement engineering during the specification of properties provides us with an easy way of establishing the requirements that the system must satisfy. It allows us to easily establish that they are the properties to be checked by the verification software.

Thus intertwining these two techniques, model checking and goal-oriented requirement engineering, enhances the software and hardware testing by introducing to it mathematic rigor and a proper requirements description, respectively.

As future work, we are working on the application of these techniques to other fields like Web Services in the Internet application world as mentioned in [10] and [9].

References


