Testing conformance of real-time software by automatic generation of observers*

Saddek Bensalem Marius Bozga Moez Krichen Stavros Tripakis
VERIMAG
Centre Equation, 2, avenue de Vignate, 38610 Gières, France. www-verimag.imag.fr.

December 22, 2003

Abstract

We propose a methodology for testing conformance of an important class of real-time applications in an automatic way. The class includes all applications for which a specification is available and can be translated into a network of timed automata. The method relies on the automatic generation of an observer from the specification, on one hand, and on the instrumentation of the system to be tested, on the other hand. The testing process consists in feeding the traces generated by the instrumented system to the observer, which is a testing device, used to check conformance of a trace to the specification. We have validated the approach on the NASA K9Rover case study.

1 Introduction

Computer-aided verification of programs has been studied for decades by the formal method research community. Different models and specification languages have been proposed to describe systems and express desired properties about them, in a precise way. The expressivity and the applicability of such models to various domains has been studied. It has been realized quite early, however, that the approach suffers from two fundamental problems of intractability. First, undecidability, because of Turing-machine expressiveness of many infinite-state models. Second, intractability because of *state-explosion*, that is, prohibitively large state spaces to be explored. A large effort then concentrated in tackling these problems, resulting in a number of significant advances. Powerful theorem-proving techniques, (semi-)automatic abstractions, symbolic representations of state space, on-the-fly algorithms, compositional and assume-guarantee methods, etc. Despite these, intractability remains a major obstacle to the applicability of formal verification.

Another major obstacle is the fact that for a number of systems, a formal models simply does not exist and is too difficult or costly to build. This is especially true in the case of software verification, where semantics for programming languages such as C or C++ do not exist.

A complementary or alternative approach, widely used in the industry today, is testing. Testing is less ambitious than verification, in the sense that it aims only at finding bugs, and not at proving correctness. Indeed, most test methods are not complete (i.e., the system cannot guaranteed to be correct even if it passes all tests). Nevertheless, confidence in the correctness of the system increases as the number of successful tests increases [26]. This feature of testing is particularly appealing to the industry, because it allows engineers to decide how much effort to put in validation, in contrast to an "all-or-nothing" verification approach.

Moreover, a model of the system under test (SUT) is often not necessary for testing. In fact, most testing methods are "black box", in the sense that the only knowledge about the SUT is its interface to the outside world (set of inputs and outputs). A model of the specification is necessary for automated test generation, however, this model is usually finite state and much smaller than a model of the SUT. This, and the feature above, makes testing tractable.

*Work partially supported by European IST projects "Next TTA" under project No IST-2001-32111 and "RISE" under project No IST-2001-38117, and by CNRS STIC project "CORTOS".

In this paper, we propose a new methodology of dynamic testing for real-time applications. It is dynamic in the sense that it makes use of instrumentation of the SUT and of run-time verification technology. The class of systems we are targetting includes all systems where a specification exists and can be translated into (or given directly as) a network of timed automata (TA) [2]. Many instances of such systems can be found in the domain of robotics. There, a *plan* defines the steps to be performed to achieve a mission, and also gives detailed information about order, timing, etc., of these steps. The plan is fed as an input to an *execution platform* (the term includes software, middleware and hardware) which must implement it, by performing the specified steps in the specified timing and order. Thus, the plan can be taken to be the specification and the platform executing this plan the SUT.

Our methodology is illustrated in Figure 1 and is described in detail in Section 2. Let us briefly summarize it here. The starting point is a plan, which is taken to be a high-level specification. This plan is automatically translated into a TA model. From the latter an *observer* is automatically synthesized. The observer is also a testing device, that is, it checks whether a sequence of observations (with time-stamps) conforms to the specification. The execution platform is instrumented, so that it can be interfaced with the observer. This interface must essentially export observable events and time-stamps for these events. The final step is the testing itself. It can be done: (1) either on-the-fly, by running the plan on the instrumented execution platform and feeding the observations to the observer; (2) or off-line, by generating a set of "log-traces" from the execution platform, then feeding these traces to the observer one by one.

The main advantage of our method is that it is potentially fully-automatic. Plans can be automatically translated into networks of TA [1]. Observers for TA can be generated automatically, as we show here. For the case study reported in this paper, we relied on the help of Klaus Havelund and Rich Washington at NASA, for the instrumentation of the execution platform and the generation of the traces. However, it should be possible to automate this part as well, in the general case, by identifying a mapping between platform events and specification events, and automatically scanning the code, adding event/time-stamp exporting commands to the identified platform events. Finally, the observation/testing process is automatic as well.

The rest of the paper is organized as follows. Section 2 presents the methodology in detail. Section 3 gives a short review of the model of timed automata. Section 4 describes plans and their translation to networks of TA. Section 5 shows how observers can be generated automatically from TA. Section 6 discusses the application of our method on the K9Rover case study. Section 7 contains conclusions and plans for future work.

Related work

[3] report work very much related to ours. Their scheme is also based on the instrumentation of the SUT and the runtime analysis of the instrumented SUT using an observer. The starting point of their method is a test-input generator, which generates inputs to the instrumented SUT. These inputs are also fed to a property generator, which generates properties that the SUT must satisfy on these particular inputs. The properties and the execution traces are fed to an observer, which checks whether the former are satisfied by the latter. The test-input generator and the property generator are specifically written for the application to be tested, while the instrumentation package and the observer are generic tools used on different applications. In one of the two case studies reported in [3], namely, the K9 rover controller, the inputs are plans like the ones we use in this paper (see Section 6). The test-input generator generates all possible plans up to given number of nodes and bounds on timing constraints.

The differences between our work and the work of [3] are as follows:

- [3] include a test-input generator and an instrumentation package in their tool-chain. Our work is still incomplete in these aspects. For the K9 rover case study, we have relied on NASA personnel and tools for the input plans, instrumentation and generation of traces.
- In [3], a set of untimed temporal logic properties are automatically generated from each plan (recently, the work has been extended to real-time temporal-logic [6]). As stated in [3], "property generation is the difficult step in [the] process" and "[the] set of properties does not fully represent the semantics of the plan, but the approach appearred to be sufficient to catch a large amount of bugs".

In our work, plans are translated into networks of timed automata. This is a fully-automatic and efficient process, which captures the full semantics of a plan [1]. Notice that, once generated, the TA corresponding to a plan can be also used for other purposes than generating an observer. For instance, to check whether the plan meets certain properties, measure delays of various sub-stages, and so on.

- The observer tools used in [3, 6] (DBRover [12], JPax [14, 7], Eagle [5, 4]), are generic tools. In our work, we automatically generate an observer for each plan. This has the potential of optimizing the observer for the particular plan.

In conclusion, we believe that our work represents an alternative that is worth pursuing.

Acknowledgments

We would like to thank Klaus Havelund and Rich Washington from NASA for their help with the instrumentation of the K9Rover application. Also, Dimitra Giannakopoulou from NASA for explaining the application.

2 Methodology

Our methodology is mainly focused at testing robotic applications, such as the NASA K9Rover (see Section 6). Such applications are often structured in two layers. A high-level *planning* layer and a low-level *execution* layer. The planning layer follows an input *plan*, which is a detailed description of the steps needed to accomplish the mission at hand. The planning layer issues commands to the execution layer, which tries to implement them and returns the results, including status information about success or failure. The planning layer then plans the next steps depending on this feedback and the instructions in the plan.

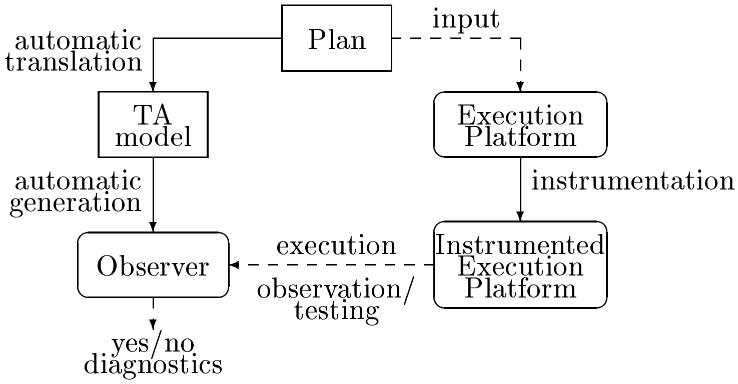

Figure 1: Methodology

A number of planning languages for robotic applications exist, see, for instance [19, 16, 17, 23, 21, 20, 13]. These languages allow to specify the ordering of the steps, their timing, how to handle exceptions or failures, and so on. Thus, they can be seen as the specification of the mission. A correct execution platform must then meet this specification. Our objective is to check this by testing. More precisely, our methodology is illustrated in Figure 1. It consists of the following phases:

1. Automatic generation of a timed-automaton specification from the plan.

2. Automatic generation of an observer from the timed-automaton specification.

3. Instrumentation of the system under test, that is, the execution platform.

4. Execution and testing of the instrumented execution platform.

We now elaborate on each of the phases.

The first step is to translate the plan in the form of a timed automaton, or a network of timed automata (TA). The translation must preserve the semantics of the plan, that is, the semantics of the TA and of the plan must be equivalent. It may also be the case that the TA *defines* the semantics of the plan in a formal way, as in [1].

Having obtained the TA specification A, the next step consists in generating automatically an *observer* for A. The observer is a testing device. It observes the system under test (SUT) and checks whether the trace generated by the SUT conforms to A. The observed traces are sequences of observable events and associated time-stamps. The accuracy of the time-stamps depends on the accuracy of the clocks of the observer. In this paper, we consider two types of observers (we follow the terminology of [15]). *Analog* observers, which can observe real-time precisely, and *digital* (or *periodic-sampling*) observers, which measure time with a clock ticking at a given period.

An observed trace conforms to A if it can be possibly generated by A. Notice that A is typically modeled as a network of TA, which induces non-determinism and internal communication between the automata. These are artifacts of the model, irrelevant to the external behavior and to the specification itself. Thus, we "hide" them, by considering them as unobservable events. This means that the observer checks if the observed trace is a possible observation resulting from some trace of A.

The third step is the instrumentation of the execution platform. It aims at interfacing the latter with the testing device (the observer). Two possibilities exist here, depending on whether the testing is to be performed *on-the-fly* (that is, during execution of the platform, which is connected to the observer at real-time) or *off-line* (that is, by first executing the platform multiple times to obtain a set of *log-traces*, then feeding these traces to the observer). In both cases, the instrumented platform must be able to expose a set of observable events to the observer. In the case of off-line testing, it must also record the time-stamps of these events. Instrumentation can be done manually or automatically. Depending on the complexity of the SUT, it can be a non-trivial task. Care should be taken so that the instrumentation does not itself alter the behavior of the system. For instance, the overhead of added code should be minimal, so as not to affect execution times of the tasks in the system. These are problems inherent in any instrumentation process, and are beyond the scope of this paper.

The final step is the testing procedure per-se. The traces generated by the instrumented platform are fed to the observer, either in real-time (for on-the-fly testing) or off-line. The observer checks conformance of each trace. If a trace is found non-conforming to the specification, the SUT is non-conforming. Otherwise, no conclusion can be made. However, confidence to the correctness of the SUT is increased with the number of tests. Obtaining a "representative" set of tests, so that some "coverage" criterion is met is a general issue in any testing method, which is beyong the scope of the present paper.

3 Preliminaries

Timed sequences, projections and digitizations

Let N be the set of non-negative integers. Let R be the set of non-negative rational numbers.

Consider a finite set of *actions* Σ. $\mathsf{RT}(\Sigma)$ (resp., $\mathsf{DT}(\Sigma)$) is defined to be the set of all finite-length *real-time sequences* (resp., *discrete-time sequences*) over Σ, that is, sequences of the form $(a_1, t_1) \cdots (a_n, t_n)$, where $n \geq 0$, for all $1 \leq i \leq n$, $a_i \in \Sigma$ and $t_i \in \mathsf{R}$ (resp., $t_i \in \mathsf{N}$), and for all $1 \leq i < j \leq n$, $t_i \leq t_j$. ϵ will denote the empty sequence. t_i will be called the *time-stamp* of a_i. Notice that time-stamps are relative to the beginning of a sequence. Thus, when concatenating sequences, they need to be adjusted. More precisely, given $\rho = (a_1, t_1) \cdots (a_n, t_n)$ and $\sigma = (b_1, t_1') \cdots (b_m, t_n')$, $\rho \cdot \sigma$ is the sequence $(a_1, t_1) \cdots (a_n, t_n)(b_1, t_n + t_1') \cdots (b_m, t_n + t_m')$. The time spent in a sequence ρ, denoted $\mathsf{time}(\rho)$, is the time-stamp of the last action (zero if the sequence is empty). For example, $\mathsf{time}((a, 0.1)(b, 1.2)) = 1.2$.

Given $\Sigma' \subseteq \Sigma$ and $\rho \in \mathsf{RT}(\Sigma)$ (resp., $\mathsf{DT}(\Sigma)$), the *projection* of ρ to Σ', denoted $P_{\Sigma'}(\rho)$, is a sequence in $\mathsf{RT}(\Sigma')$ (resp., $\mathsf{DT}(\Sigma')$), obtained by "erasing" from ρ all pairs (a_i, t_i) such that $a_i \notin \Sigma'$. For example, if $\Sigma = \{a, b\}$, $\Sigma' = \{a\}$ and $\rho = (a, 0)(b, 1)(a, 3)$, then $P_{\Sigma'}(\rho) = (a, 0)(a, 3)$. For a set of sequences $L \subseteq \mathsf{RT}(\Sigma)$ (or $L \subseteq \mathsf{DT}(\Sigma)$), $P_{\Sigma_{obs}}(L) = \{P_{\Sigma_{obs}}(\rho) \mid \rho \in L\}$.

Consider $\delta \in \mathsf{R}$, $\delta > 0$, and $\rho \in \mathsf{RT}(\Sigma)$. The *digitization* of ρ with respect to δ, denoted $[\rho]_\delta$, is a sequence in $\mathsf{DT}(\Sigma)$, obtained by replacing every pair (a_i, t_i) in ρ by $(a_i, \lfloor \frac{t_i}{\delta} \rfloor)$, where $\lfloor x \rfloor$ is the integral part of x. For example, if $\rho = (a, 0.1)(b, 0.9)(c, 1)(d, 2.3)$, then $[\rho]_1 = (a, 0)(b, 0)(c, 1)(d, 2)$ and $[\rho]_{0.5} = (a, 0)(b, 1)(c, 2)(d, 4)$. For a set of sequences $L \subseteq \mathsf{RT}(\Sigma)$, $[L]_\delta = \{[\rho]_\delta \mid \rho \in L\}$.

Timed automata

We use timed automata (TA) [2] with *deadlines* to model urgency [22, 8]. A *timed automaton over* Σ (TA) is a tuple $A = (Q, q_0, X, \Sigma, \mathsf{E})$ where Q is a finite set of *locations*; $q_0 \in Q$ is the initial location; X is a finite set of *clocks*; E is a finite set of *edges*. Each edge is a tuple (q, q', ψ, r, d, a), where $q, q' \in Q$ are the source and destination locations; ψ is the *guard*, a conjunction of constraints of the form $x \# c$ or $x - y \# c$, where $x, y \in X$, c is an integer constant and $\# \in \{<, \leq, =, \geq, >\}$; $r : X \to \{0\} \cup X$ is the clock *reset* function; $d \in \{\mathsf{lazy}, \mathsf{delayable}, \mathsf{eager}\}$ is the *deadline*; and $a \in \Sigma$ is the action. We will not allow eager edges with guards of the form $x > c$.

A TA A defines an infinite labeled transition system (LTS). Its states are pairs $s = (q, v)$, where $q \in Q$ and $v : X \to \mathsf{R}$ is a clock *valuation*. $\vec{0}$ is the valuation assigning 0 to every clock of A. S_A is the set of all states and $s_0^A = (q_0, \vec{0})$ is the initial state. There are two types of transitions. Discrete transitions of the form $(q, v) \xrightarrow{a}_A (q', v')$, where $a \in \Sigma$ and there is an edge (q, q', ψ, r, d, a), such that v satisfies ψ and v' is obtained by resetting to zero all clocks in r and leaving the others unchanged. Timed transitions of the form $(q, v) \xrightarrow{t}_A (q, v + t)$, where $t \in \mathsf{R}, t > 0$ and there is no edge (q, q'', ψ, r, d, a), such that: either $d = \mathsf{delayable}$, $v \models \psi$ and $v' \not\models \psi$; or $d = \mathsf{eager}$ and $v \models \psi$.

We use notation such as $s \xrightarrow{a}_A$, $s \not\xrightarrow{a}_A$, ..., to denote that there exists s' such that $s \xrightarrow{a}_A s'$, there is no such s', and so on. This notation extends to sequences in $\mathsf{RT}(\Sigma)$: $s \xrightarrow{\epsilon}_A s$ and if $s \xrightarrow{\rho}_A s'$ and $s' \xrightarrow{t}_A \xrightarrow{a}_A s''$, then $s \xrightarrow{\rho'}_A s''$, where $\rho' = \rho \cdot (a, \mathsf{time}(\rho) + t)$.

A state $s \in S_A$ is *reachable* if there exists $\rho \in \mathsf{RT}(\Sigma)$ such that $s_0^A \xrightarrow{\rho}_A s$. The set of reachable states of A is denoted $\mathsf{Reach}(A)$.

The set of *traces* of a TA A over Σ is defined to be

$$\mathsf{Traces}(A) = \{\rho \in \mathsf{RT}(\Sigma) \mid s_0^A \xrightarrow{\rho}_A\}. \tag{1}$$

Let $\Sigma_{obs} \subseteq \Sigma$ be a set of *observable* actions. The actions in $\Sigma \setminus \Sigma_{obs}$ are called *unobservable*. The set of *observed traces* of A with respect to Σ_{obs} is defined to be

$$\mathsf{ObsTraces}(A, \Sigma_{obs}) = P_{\Sigma_{obs}}(\mathsf{Traces}(A)). \tag{2}$$

Given $\delta \in \mathsf{R}$, $\delta > 0$, the set of *δ-digital observed traces* of a TA A is defined to be

$$\mathsf{DigTraces}(A, \Sigma_{obs}, \delta) = [\mathsf{ObsTraces}(A, \Sigma_{obs})]_\delta. \tag{3}$$

Notice that $\mathsf{Traces}(A) \subseteq \mathsf{RT}(\Sigma)$, $\mathsf{ObsTraces}(A, \Sigma_{obs}) \subseteq \mathsf{RT}(\Sigma_{obs})$ and $\mathsf{DigTraces}(A, \Sigma_{obs}, \delta) \subseteq \mathsf{DT}(\Sigma_{obs})$.

4 Generating timed-automata from plans

In this section we describe how to obtain TA models from plans. We give the construction for the concrete language of plans performed by the K9 Rover executive, which is actually our case study (see section 6). Nevertheless, TA models are general enough to capture most of the constraints expressed in plan languages.

For the K9 Rover application, a plan is a hierarchical structure of actions that the executive must perform. Traditionally, plans are deterministic sequences of actions. However, increased autonomy requires added flexibility. The plan language therefore allows branching based on conditions that need to be checked, and also for flexibility with respect to the starting time of an action. We give here an example of a language used in the description of the plans that the executive must execute.

Plan	$\longrightarrow$	*Node*
Node	$\longrightarrow$	*Block* | *Task*
Block	$\longrightarrow$	(**block**
		NodeAttr
		:node-list (*Node ... Node*))
Task	$\longrightarrow$	(**task**
		NodeAttr
		:action *Symbol*)
NodeAttr	$\longrightarrow$	**:id** *Symbol*
		:start-condition *Condition*
		:waitfor-condition *Condition*
		:maintain-condition *Condition*
		:end-condition *Condition*
		[**:continue-on-failure**]
Condition	$\longrightarrow$	(**time** *StartTime EndTime*)

Figure 2: Plan concrete grammar

Plan Syntax

A plan is a node, a node is either a task, corresponding to an action to be executed, or a block, corresponding to a logical group of nodes. Figure 4 shows the grammar for the language that we considered to describe plans; we should note that all the node attributes, with the exception of the node's id, are optional. Each node may specify a set of *conditions*, e.g., the *start condition* (that must be true at the beginning of the node execution), the *wait for conditions* (wait for if the condition is not true), the maintain condition (must be true through the execution of the node) and the *end condition* (that must be true at the end of the node execution). Each condition includes information about relative or absolute time window, indicating a lower and upper bound on the time. The *continue-on-failure* flag indicates what the behavior will be when node failure is encountered.

We propose hereafter a compilation method allowing to obtain from a plan (that is, a syntactic object) a network of timed automata (that is, a semantic model) encoding all the accepted, reasonable executions of that plan.

For sake of simplificity, we consider the following abstract syntax for plans.

Definition 4.1 (plan syntax)
A plan P is a tuple $(N, \delta, \lambda, n_0)$ where

- *N is a finite set of nodes*
- *$\delta : N \to N^*$ is the node decomposition function, defined such that the image set relation $\hat{\delta} = \{(n, n') | n' \in \delta(n)\}$ satisfies*
  - *acyclicity: $\forall n \in N.\ \ n \notin \hat{\delta}^+(\{n\})$*
  - *disjointness: $\forall n_1, n_2 \in N, n_1 \neq n_2.\ \ \hat{\delta}^+(\{n_1\}) \cap \hat{\delta}^+(\{n_2\}) = \emptyset$*
- *$\lambda : N \to \Sigma \times \mathcal{I}^4 \times \mathcal{B}$ is the node labeling function, where Σ is a set of action labels, $\mathcal{I} = \{[l, u] \mid l, u \in \mathbb{N}\}$ is the set of interval constraints, and $\mathcal{B}$ are the booleans. That is, $\lambda(n) = (a_n, (s_n, w_n, m_n, e_n), f_n)$ where a_n is the action symbol, s_n, w_n, m_n, e_n are respectively the start, wait-for, maintain and end timed constraints, and f_n is the continue-on-failure flag associated to the node n.*
- *$n_0 \in N$ is the main (or start) node of the plan*

Plan Semantics

For every node, execution proceeds through the following steps given :

1. Wait until the start condition is satisfied; if the current time passes the end of the start condition, the node times out and this is a node failure.

2. The execution of a *task* proceeds by invoking the corresponding action. The action takes exactly the time specified in the **:duration** attribute. The action's status must be fail, if **:fail** is true or the time conditions are not met; otherwise, the status must be success. The execution of a block simply proceeds by executing each of the node in the node-list in order.

3. If the time exceeds the end condition, the node fails.

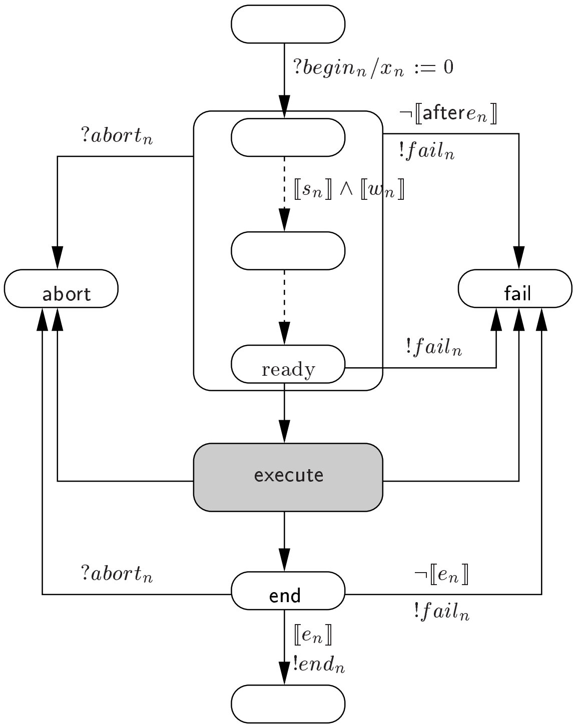

Figure 3: Timed automaton for the common part

On a *node failure* occuring in a sequence, the value of the enclosing block node's *continue-on-failure* flag is checked. If true, execution proceeds to the next node in the sequence. If false, the node failure is propagated to the block enclosing node and so on. If the node failure passes out to the top level block of the plan, the execution is aborted.

We present now the semantics of nodes and plans in terms of timed automata. The semantics is *constructive* in the sense that, automata can be effectively constructed, depending on syntactical description of the nodes. The semantics is also *compositional* in the sense that, the semantics of the plan is obtained directly by composition of timed automata associated to nodes.

Let first introduce some notations for some given plan $P = (N, \delta, \lambda, n_0)$. The set of actions Σ_P contains respectively the set of synchronisation actions $begin_n$, $abort_n$, $fail_n$, end_n defined for all nodes, and the set of elementary actions a_n, defined for task nodes of the plan.

The set of clocks $X_P = \{x_n \mid n \in N\}$ contains one clock x_n for each node n of the plan. This clock x_n is set to 0 when the execution of the node n begins. If $c_n = [l, u]$ is some constraint of the node n, we will note with $[\![c]\!]$ the timed guard $(l \leq x_n \wedge x_n \leq u)$. We note also with $\mathsf{after} c$ the constraint $[-\infty, u]$ where the lower bound of c has been removed.

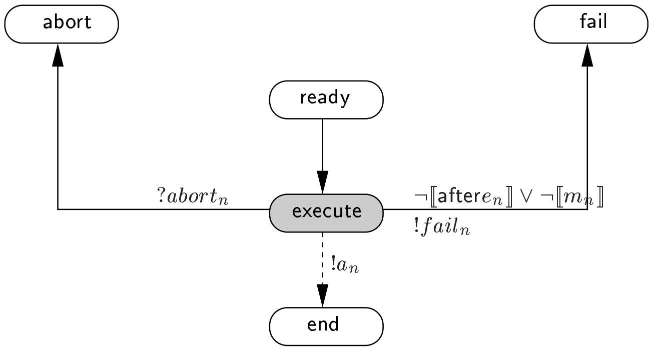

Figure 4: Timed automaton for the task-specific part

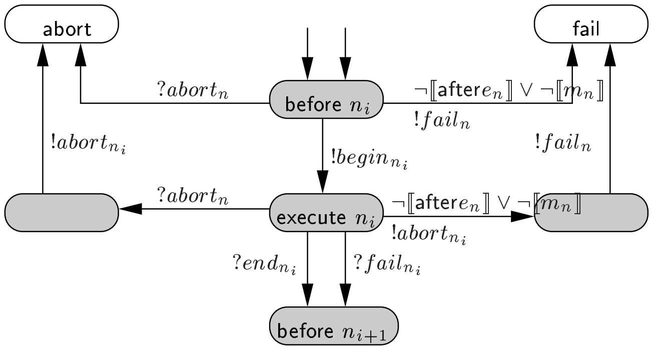

Figure 5: Timed automaton for the block-specific part with continue-on-failure

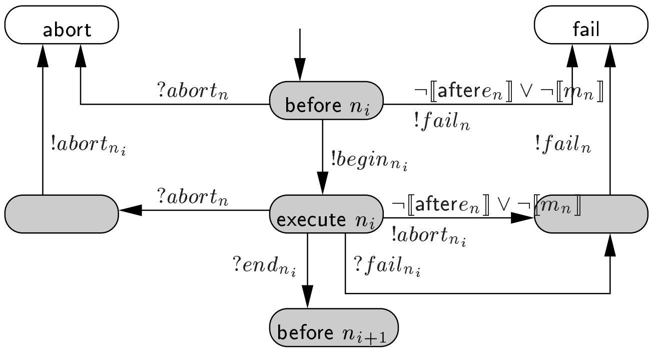

Figure 6: Timed automaton for the block-specific part without continue-on-failure

To each node n of P we associate a timed automaton over clocks X_P and actions Σ_P. The automaton encode the sequential behaviour described by the node execution algorithm. Note that, since the execution algorithm is deterministic, the timed automata obtained are determnistic also.

Definition 4.2 (node semantics) *Let n be a node with $\delta(n) = n_1...n_k$ and $\lambda(n) = (a_n, (s_n, w_n, m_n, e_n), f_n)$. The semantics of the node n is described by the timed automaton from figure 3. The specific part is filled* *according to node attributes as follows:*

- *task node ($\delta(n) = \epsilon$) see figure 4*
- *block node ($\delta(n) = n_1...n_k$) with continue on failure ($f_n = true$) see figure 5*
- *block node ($\delta(n) = n_1...n_k$) without continue on failure ($f_n = false$) see figure 6*

Finally, the semantics of the entire plan P is given by the parallel composition i.e, the network of timed automata defined for all of its nodes. Note that, the product automaton is deterministic too.

Definition 4.3 (plan semantics)
Let $P = (N, \delta, \lambda, n_0)$ be a plan, X_P the set of clocks and Σ_P the set of actions defined by P. Let A_{n_i} be the timed automata over X_P and Σ_P associated to nodes $n_i \in N$. The semantics of the plan P is given by the network $A_{n_0}||A_{n_1}||...||A_{n_k}$.

An example of a plan and the corresponding timed automata are given in Figure 7.

5 Generating observers from timed-automata specifications

In this section, we define two kinds of observers for a TA A. They are distinguished by their observation capabilities with respect to time. The first, called analog observers (the terminology is taken from []) can observe a set of observable actions and the exact time-stamps of these actions. Thus, these observers can be thought of possessing a "perfect", real-time clock, which they can consult immediately upon observing an action. The second, called digital observers (or periodic-sampling observers) can also observe a set of observable actions, but they only dispose of a digital clock, that is, a counter that ticks with a given period δ. Thus, when observing an action a which occurred at real-time t, the digital observer only knows the current value of its periodic clock, i.e., $\lfloor \frac{t}{\delta} \rfloor$.

The objective of the observers is to determine whether a given trace (generated by a system under observation) could be possibly generated by A. If it could, then the system under observation passes this test, otherwise, it fails. Thus, A can be seen as the specification for the system under observation.

It is worth pointing out that, in case A is deterministic and fully observable (that is, all actions are observable), an analog observer performs nothing else but a membership test: does the observation belong to the language generated by A. (Notice that A has no acceptance conditions in our setting, thus, languages are prefix-closed.)

Analog and digital observer definition

Let us formalize the above notions. Let A be a TA over Σ and let $\Sigma_{obs} \subseteq \Sigma$ be a set of observable actions.

An *analog observer* for A with respect to Σ_{obs} is a total function

$$\mathsf{O} : \mathsf{RT}(\Sigma_{obs}) \to \{0, 1\} \tag{4}$$

such that

$$\forall \rho \in \mathsf{RT}(\Sigma_{obs}) \ . \ \Big(\mathsf{O}(\rho) = 1 \Leftrightarrow \rho \in \mathsf{ObsTraces}(A, \Sigma_{obs})\Big). \tag{5}$$

In words, O returns 1 iff ρ is an observed trace of A.

A *digital observer* (or periodic-sampling observer) for A with respect to Σ_{obs} and $\delta > 0$ is a total function

$$\mathsf{D} : \mathsf{DT}(\Sigma_{obs}) \to \{0, 1\} \tag{6}$$

such that

$$\forall \rho \in \mathsf{DT}(\Sigma_{obs}) \ . \ \Big(\mathsf{D}(\rho) = 1 \Leftrightarrow \rho \in \mathsf{DigTraces}(A, \Sigma_{obs}, \delta)\Big). \tag{7}$$

In words, D returns 1 iff ρ is a δ-digital observed trace of A.

```
(block
  :id node0
  :continue-on-failure T
  :start-temporal-conditions ((1 5))
  :end-temporal-conditions ((1 30))
     :node-list (
        (block
          :id node1
          :continue-on-failure T
          :start-temporal-conditions (( node0 start (1 5)))
        )
        (block
          :id node2
          :continue-on-failure T
          :start-temporal-conditions (( node1 end (1 5)))
          :end-temporal-conditions (( node2 start (1 30)))
        )
     )
)
```

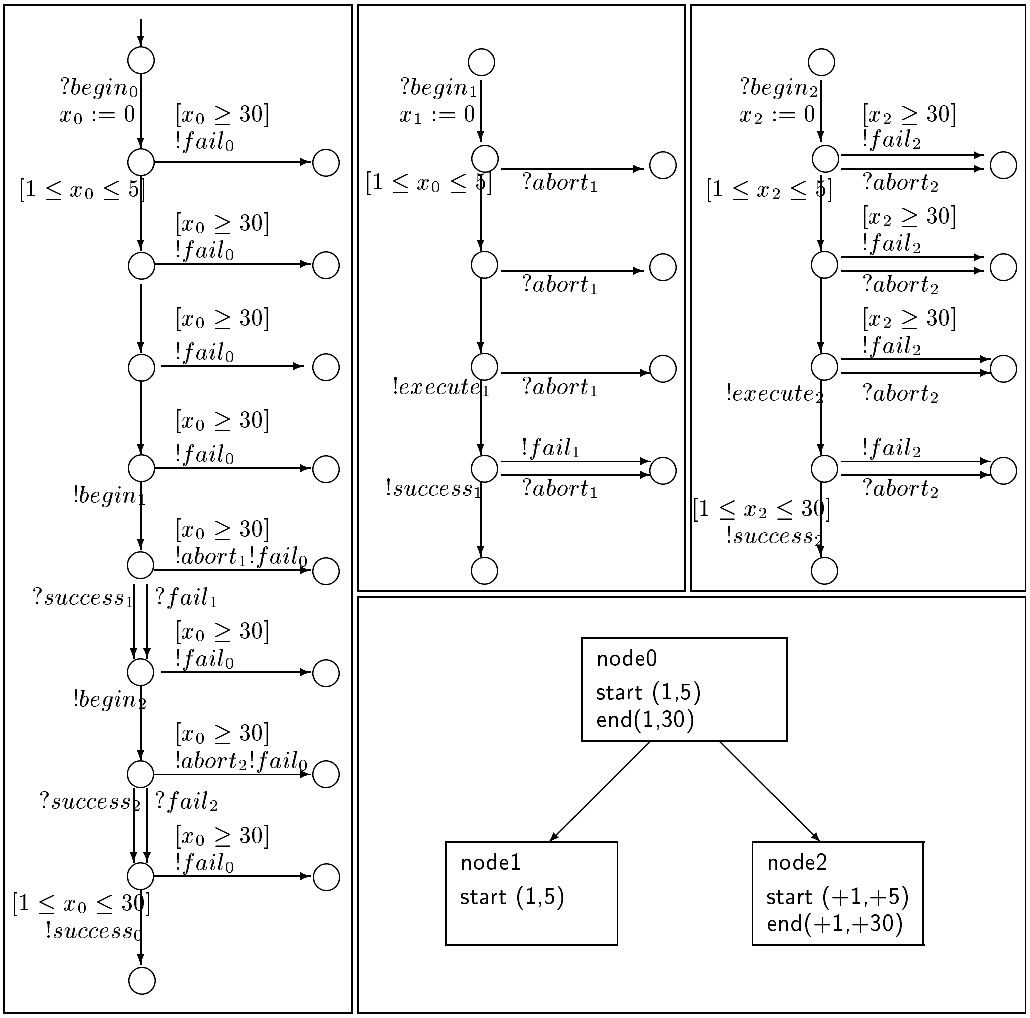

Figure 7: A plan and its translation to a network of three timed automata.

Automatic observer generation using the state-estimation technique

We next show how, given a timed automaton A, analog and digital observers can be automatically generated for A. The method relies on the *state-estimation* technique proposed in [24], where it was applied to fault detection.

State estimation consists in computing, given an observation, the set of states of A which "match" this observation, that is, the set of all possible states which can be reached by some trace which yields the observed trace. If the state estimate remains non-empty all along the observation, then the latter can indeed be generated by A, since there exists at least one trace of A matching the observation. If, however, the state estimate becomes empty, then the observation cannot be generated by A.

State estimation is not more expensive than reachability analysis. In fact, in some cases (deterministic, fully observable specification), it is much cheaper, as stated above. As shown in [24, 18], state estimates can be represented using standard data structures for TA, such as DBMs [11], and can be computed using various versions of symbolic successor operators, depending on the desired estimator (analog or digital).

Generating analog observers

Consider a timed automaton A over Σ and a set of observable actions $\Sigma_{obs} \subseteq \Sigma$. The *analog state-estimator* for A with respect to Σ_{obs} is the total function $\mathsf{SE_a} : \mathsf{RT}(\Sigma_{obs}) \to 2^{\mathsf{Reach}(A)}$, defined as follows:

$$\mathsf{SE_a}(\rho) \quad = \quad \{s \mid \exists \sigma \in \mathsf{RT}(\Sigma) \,.\, s_0^A \xrightarrow{\sigma}_A s \wedge P_{\Sigma_{obs}}(\sigma) = \rho\}. \tag{8}$$

$\mathsf{SE_a}(\rho)$ contains all states where A can possibly be after executing a sequence which yields the analog observation ρ.

Now, define

$$\mathsf{O}(\rho) = \begin{cases} 1, & \text{if } \mathsf{SE_a}(\rho) \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

It follows easily from the definitions that O defined as above is a valid analog observer for A w.r.t. Σ_{obs}.

We proceed to discuss how $\mathsf{SE_a}$ can be computed. Let S be a set of states of A. Let $a \in \Sigma$ and $t \in \mathsf{R}$. Define the following operators:

$$\begin{aligned} \mathsf{asucc}(S,a) \quad &= \quad \{s' \mid \exists s \in S \,.\, s \xrightarrow{a}_A s'\} & (10)\\ \mathsf{tsucc}(S,t) \quad &= \quad \{s' \mid \exists s \in S \,.\, \exists \rho \in \mathsf{RT}(\Sigma \setminus \Sigma_{obs}) \,.\, \mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho}_A s'\} & (11) \end{aligned}$$

$\mathsf{asucc}(S,a)$ contains all states which can be reached by some state in S after performing action a. $\mathsf{tsucc}(S,t)$ contains all states which can be reached by some state in S via a sequence ρ which contains no observable actions and takes exactly t time units.

The following proposition shows how $\mathsf{SE_a}(\rho)$ can be computed recursively on ρ.

Proposition 5.1

$$\begin{aligned} \mathsf{SE_a}(\epsilon) \quad &= \quad \mathsf{tsucc}(\{s_0^A\}, 0) & (12)\\ \mathsf{SE_a}(\rho \cdot (a,t)) \quad &= \quad \mathsf{asucc}(\mathsf{tsucc}(\mathsf{SE_a}(\rho), t - \mathsf{time}(\rho)), a) & (13) \end{aligned}$$

Generating digital (periodic-sampling) observers

Consider a timed automaton A over Σ and a set of observable actions $\Sigma_{obs} \subseteq \Sigma$. Let $\delta \in \mathsf{R}$, $\delta > 0$. The *digital state-estimator* for A with respect to Σ_{obs} and δ is the total function $\mathsf{SE_d} : \mathsf{DT}(\Sigma_{obs}) \to 2^{\mathsf{Reach}(A)}$, defined as follows:

$$\mathsf{SE_d}(\rho) \quad = \quad \{s \mid \exists \sigma \in \mathsf{RT}(\Sigma) \,.\, s_0^A \xrightarrow{\sigma}_A s \wedge [P_{\Sigma_{obs}}(\sigma)]_\delta = \rho\}. \tag{14}$$

$\mathsf{SE_d}(\rho)$ contains all states where A can possibly be after executing a sequence which yields the digital observation ρ.

Now, define

$$\mathsf{D}(\rho) = \begin{cases} 1, & \text{if } \mathsf{SE_d}(\rho) \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{15}$$

It follows easily from the definitions that D defined as above is a valid digital observer for A w.r.t. Σ_{obs} and δ.

We proceed to discuss how D can be computed. We first form the product $A_{\mathsf{tick}} = A || \mathsf{Tick}$ (we assume that $\mathsf{tick} \notin \Sigma$). Let S be a set of states of A_{tick}. Define the following operator:

$$\mathsf{usucc}(S) = \{s' \mid \exists s \in S \;.\; \exists \rho \in \mathsf{RT}(\Sigma \setminus \Sigma_{obs}) \;.\; s \xrightarrow{\rho}_{A_{\mathsf{tick}}} s'\}. \tag{16}$$

$\mathsf{usucc}(S)$ contains all states which can be reached by some state in S via a sequence ρ which contains no observable actions. Notice that, by construction of A_{tick}, the duration of ρ is bounded: since tick is observable and has to occur after at most 1 time unit, $\mathsf{time}(\rho) \leq 1$.

For $a \in \Sigma \cup \{\mathsf{tick}\}$, define:

$$\mathsf{dsucc}(S, a) = \mathsf{asucc}(\mathsf{usucc}(S), a). \tag{17}$$

As usual, $\mathsf{dsucc}^k(\cdot, a)$, for $k \in \mathsf{N}$, denotes the application of $\mathsf{dsucc}(\cdot, a)$ k times. That is, $\mathsf{dsucc}^0(\cdot, a)$ is the identity function and $\mathsf{dsucc}^{k+1}(\cdot, a) = \mathsf{dsucc}(\mathsf{dsucc}^k(\cdot, a), a)$.

The following proposition shows how $\mathsf{SE_d}(\rho)$ can be computed recursively on ρ.

Proposition 5.2

$$\mathsf{SE_d}(\epsilon) = \{s_0^{A_{\mathsf{tick}}}\} \tag{18}$$

$$\mathsf{SE_d}(\rho \cdot (a, k)) = \mathsf{dsucc}(\mathsf{dsucc}^k(S, \mathsf{tick}), a) \tag{19}$$

Implementation

We have implemented a prototype observer generation tool, called If2Obs, on top of the IF environment [9]. The IF modeling language allows to specify systems consisting of many processes communicating through message passing or shared variables and includes features such as hierarchy, priorities, dynamic creation and complex data types. The IF tool-suite includes a variety of tools for simulation, model checking and test generation. If2Obs is written in C++ and uses the basic libraries of IF for parsing and symbolic reachability of timed automata with deadlines.

If2Obs takes as main input the specification automaton, written in IF language, and generates an off-line observer. The observer takes as input a trace and the set of unobservable actions of the original IF specification. The observer can function either as an analog observer (default) or as a digital observer (`-d` option).

6 Case Study

Our case study is the Mars rover controller K9, and in particular its executive subsystem, developed at NASA Ames. It is an experimental platform for autonomous wheeled vehicles called rovers, targeted for the exploration of the Martian surface. K9 is specifically used to test out new autonomy software, such as the Rover Executive. The Rover Executive provides a flexible means of commanding a rover through plans that control the movement, experimental apparatus, and other resources of the Rover - also taking into account the possibiliy of failure of command actions.

The Rover executive is a software prototype written in C++ by researchers at NASA Ames. It is a multi-threaded program that consists of approximately 35,000 lines of C++ code, of which 9600 lines of code are related to actual functionality. The C++ code was manually translated into Java and C to experiment with tools using three technologies: static analysis, model checking and runtime analysis [10, 3].

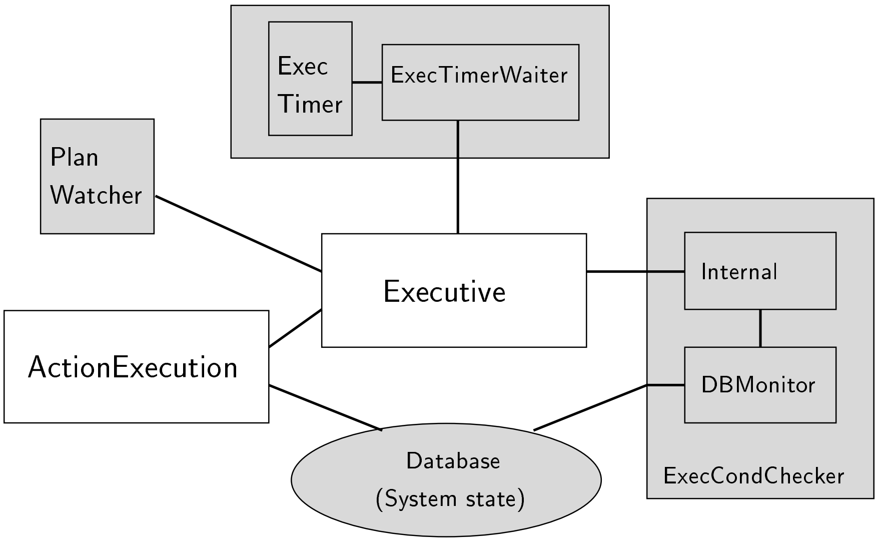

Figure 8: The K9 Rover Architecture

System Description

The Rover executive is made up of :

- A main coordinating component named Executive. It provides the main control to how the plan is executed. Executive waits for a plan to be available, and signals at the end of the plan execution. So, the PlanWatcher signals when plan is ready, and waits for end of execution to send new plan.
- The component for monitoring the state conditions ExecCondChecker consists of two threads, the database monitor DBMonitor just keeps watching for changes in the database and the thread Internal decides what needs to be done about it.
- ExecTimerChecker is the component for monitoring temporal conditions. It consists of two threads ExecTimer and ExecTimerWaiter. Both are respectively very similar to DBMonitor and Internal
- The ActionExecution thread is responsible for issuing the commands to the Rover. It consists of a list of methods internalDoAction, doAction, stopAction, and abortAction. T ActionExecution runs the internalDoAction method, the other methods are just called by the Executive on its own thread by simple calls.
- Database simply receives calls to its methods. All accesses to the database through its methods are controlled by a lock.

Synchronization between these threads is performed through mutex and conditions variables.

Results

Due to intellectual property restrictions, we did not have access to the execution platform of the K9Rover. However, NASA provided us with a set of one hundred plans and traces, generated by the K9Rover execution platform. We have then applied our method, using the plan-to-IF translator to obtain IF models for each plan, and If2Obs to generate an observer for each IF model of a plan. The observer was then used to check the traces.

One of the plans is shown in Figure 7. Time units in the plan are in seconds. A trace generated by the execution platform with input this plan is shown below (times in the trace are in milliseconds):

```
start node0 922
start node1 1932
success node1 1932
start node2 2942
success node2 2942
success node0 2942
```

The trace says that node 0 starts at time 922, node 1 starts at time 1932 and completes successfully at the same time, and so on. This trace does not conform to the specification, because the latter requires that node 2 finishes at least 1 second after it starts (fifth line of the block of node 2 in the plan). If2Obs takes a few seconds to generate the observer (this is a C++ code generation process, compilation and linking with IF libraries). The observer less than a second to disqualify this claim as non-conforming. In general, all traces were checked in a matter of seconds.

7 Conclusions and future work

We have proposed a methodology for testing conformance of an important class of real-time applications in an automatic way. The class includes all applications for which a specification is available and can be translated into a network of timed automata. In particular, the class includes robotic applications where specifications are considered to be the plans describing the robot mission. Such plans can be automatically translated to timed automata [1].

The method relies on the automatic generation of an observer from the specification, on one hand, and on the instrumentation of the system to be tested, on the other hand. The testing process consists in feeding the traces generated by the instrumented system to the observer, which is a testing device, used to check conformance of a trace to the specification. We have validated the approach on the NASA K9Rover case study.

Regarding future work, we plan to study the instrumentation and trace generation problems. As mentioned above, instrumentation should be possible to automate, by identifying a mapping between execution platform events and specification events, and automatically scanning the code, adding event/time-stamp exporting commands to the identified platform events. Trace generation has a lot of similarities to test-case generation, since the system under test must be run a number of times, with different inputs, to obtain a set of traces. Input coverage and other techniques can be employed here to obtain an adequate, in some sense, set of traces. In fact, it is an interesting question how to define coverage in the case where the inputs are plans.

We also plan to study the representation of observers as finite automata (timed or untimed). This is not always possible, because timed automata are non-determinizable in general [2]. Moreover, checking whether a particular TA is determinizable and determinizing it is algorithmically impossible, in general [25]. Identifying classes of TA (e.g., those generated from plans) for which the above problems are solvable is a possible step in this direction.

Finally, we envisage extending our approach to other high-level specification languages and experimenting with more case studies.

References

[1] A. Akhavan, S. Bensalem, M. Bozga, and E. Orfanidou. Experiment on verification of a planetary rover controller, 2003. Submitted.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In *10th International Workshop on Abstract State Machines (ASM'03)*, March 2003.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Eagle does Space Efficient LTL Monitoring. Submitted for publication, October 2003.

[5] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Eagle Monitors by Collecting Facts and Generating Obligations. Submitted for publication, October 2003.

[6] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of Fifth International Conference on Verification, Model Checking and Abstract Interpretation, January 2004 – to appear in LNCS.*, August 2003.

[7] S. Bensalem and K. Havelund. Deadlock Analysis of Multi-threaded Java Programs. Submitted for publication, December 2003.

[8] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality*, volume 1536 of *LNCS*. Springer, 1998.

[9] M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: a validation environment for timed asynchronous systems. In E.A. Emerson and A.P. Sistla, editors, *Proc. CAV'00*, volume 1855 of *LNCS*, pages 543–547. Springer Verlag, 2000.

[10] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, and W. Visser. Experimental evaluation of v&v tools on martian rover software. In *SEI Software Model Checking Workshop*, 2003.

[11] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, pages 197–212. Springer–Verlag, 1989.

[12] D. Drunsinsky. The temporal tover and the atg rover.

[13] R. Goldman. Model-based planning and real-time execution in the CIRCA framework. In *AI Planning and Scheduling (AIPS'02)*, 2002.

[14] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Proceedings of the First International Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 97–114, Paris, France, July 2001. Elsevier Science. Extended version to appear in the journal *Formal Methods in System Design*.

[15] T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP'92*, LNCS 623, 1992.

[16] F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *IEEE Intl. Conf. on Robotics and Automation*, 1996.

[17] K. Konolige, K. L. Myers, E. H. Ruspini, and A. Saffiotti. The Saphira architecture: A design for autonomy. *Journal of experimental & theoretical artificial intelligence*, 9(1):215–235, 1997.

[18] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems, 2003. Submitted.

[19] D.M. Lyons. Representing and analyzing action plans as networks of concurrent processes. *IEEE Transactions on Robotics and Automation*, 1993.

[20] N. Muscettola, G.A. Dorais, C. Fry, R. Levinson, and C. Plaunt. IDEA: Planning at the core of autonomous reactive agents. In *AI Planning and Scheduling (AIPS'02)*, 2002.

[21] J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *IEEE Conf. on Robotics and Automation*, 1999.

[22] J. Sifakis and S. Yovine. Compositional specification of timed systems. In *13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96*, pages 347–359, Grenoble, France, February 1996. Lecture Notes in Computer Science 1046, Spinger-Verlag.

[23] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Intelligent Robotics and Systems*, 1998.

[24] S. Tripakis. Fault diagnosis for timed automata. In *Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'02)*, volume 2469 of *LNCS*. Springer, 2002.

[25] S. Tripakis. Folk theorems on the determinization and minimization of timed automata. In *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, LNCS. Springer, 2003.

[26] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4), 1997.