Experiment on Verification of a Planetary Rover Controller

Anahita Akhavan, Saddek Bensalem, Marius Bozga and **Eleni Orfanidou**
VERIMAG - Centre Equation, 2 Avenue de Vignate, 38610 Gières
{akhavan,bensalem,bozga,orfanido}@imag.fr

Abstract. In this paper, we report an experiment on the verification of the K9 Rover Executive, an experimental platform for autonomous vehicles targeted for the exploration of the Martian surface developed at NASA Ames. The Executive provides a means to control and command the vehicles through predefined plans, that are hierarchical descriptions of actions annotated with real-time constraints. The verification concerns the correctness of the Executive, which must execute the plans according to their semantics.

1 Introduction

Proof of correctness is a collection of techniques that apply the formality and rigor of mathematics to the task of proving consistency between an algorithmic solution and a rigorous, complete specification of the intent of the solution. This technique is also often referred to as ”formal verification”. The usual proof technique follows Floyd’s Method of Inductive Assertions (Floyd 1967; Hantler and King 1976). Proof of correctness techniques are usually presented in the context of verifying an implementation against a specification.

Formal verification has increasingly gained acceptance within hardware (Burch *et al.* 1990; McMillan 1993) and protocol verification (Holzmann 1991) as an additional means to discovering errors. However, there are several limitations to formal verification. One limitation has to do with verifying large systems. In general, the state space is often much bigger and the relationships harder to understand because of asynchronous behavior. Many techniques have been developed in order to push further the limits of verification technology. One of them consists in using *property preserving abstractions*: Given a program and a property to be verified, find a (simpler) abstract program such that the satisfaction on the abstract program implies the satisfaction on the initial program, called concrete program in this context. An important point is, given a concrete program, how to *construct* an abstract program that is both, simple enough in order to be verified by available tools, and that still contains enough relevant details for the satisfaction of the considered properties.

Verifying large programs written in real programming languages is different from verifying hardware or protocols: the amount of details to handle, combined with the lack of powerful tools and a more complicated underlying semantics may make the proof technique impractical. Few attemps have been made to automatically verify programs written in real programming languages. The most known attempt is Java Pathfinder (Havelund and Pressburger 2000) (JPF). JPF is an explicit-state model checker for programs written in Java. As any model checker JPF suffers from state-explosion problem and even more acute when checking actual source code.

A complementary or alternative approach widely used in the industry today is testing. Testing is less ambitious than verification, in the sense that it only aims at finding bugs, and not at proving correctness. Indeed, most test methods are not complete (i.e., the system cannot be 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 (Zhu *et al.* 1997). 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.

In (Bensalem *et al.* 2004), we proposed a new methodology of dynamic testing for real-time applications. It is dynamic in the sense that it makes use of instrumentation of the system under test (SUT) and of run-time verification technology. We applied this methodology for testing the NASA K9Rover (see Section 3).

In this paper, we report an experiment on the verification of the K9 Rover Executive, an experimental platform for autonomous vehicles targeted for the exploration of the Martian surface developed at NASA Ames. The Executive provides a means to control and command the vehicles through predefined plans, that are hierarchical descriptions of actions annotated with real-time constraints. The verification concerns the correctness of the Executive, which must execute the plans according to their semantics.

The verification approach consists of three steps. First, we extract an abstract operational model of the actual software as a composition of timed automata. It will allow us to approximate, for a given plan, *all* concrete executions of the software for that plan. Second, we give an operational semantics of plans in terms of timed automata too. That is, for a given plan, we construct a plan observer encoding the set of all *bad* executions for the plan. Third, the verification step computes the intersection of concrete executions and the bad ones i.e, by exploring the product between the abstract specification and the plan observer. Common executions correspond to errors of the Executive.

The paper is organized as follows. Section 2 recalls some basic results of timed automata theory. Section 3 introduces the case study. The verification approach and the results are presented in section 4. Finally, some conclusions and perspectives are presented in section 5.

Related work

Two different works considered the K9 Rover Executive to experiment different approaches:

- In (Artho *et al.* 2003) an experimental evaluation of verification and validation tools, by NASA Ames computer scientists, on K9 rover has been done. The objective of the study was to assess the maturity of different technologies. The experiment consisted of using specific technology to analyse the same code. The technologies used were model checking, runtime analysis, static analysis and testing. The tools associated with these technologies analyse a program for particular kinds of bugs, or rather coding errors. The static tool was the commercial Polyspace tool. It focuses on finding errors that lead to run-time faults such as underflow/overflow, non-initialized variables, null pointer de-referencing, and array bound checking. The model checking tool was Java PathFinder (JPF), which is an explicit-state model checker that works directly on Java code. The runtime analysis tools were Java PathExplorer (JPaX) and DBRover.
- The work presented in the paper (Brat *et al.* 2003) describes experiments with test case generation and runtime analysis.

2 Preliminaries

We recall here the model of timed automata with urgencies from (Bornot *et al.* 1997). Then, we recall the most important decidability results concerning timed automata in general.

2.1 Timed Automata

Let X be a finite set of real-valued variables, called *clocks*, and Σ be a finite set of action names, the alphabet. We define the set of *guards* $G(X)$ over X as being the set of clock constraints defined by the grammar $g ::= x\#k \mid x-y\#k \mid g \wedge g$, where $x, y \in X$ are clocks, $k \in \mathbb{N}$ denotes a natural number and # denotes any of the relational operators $\{<, \leq, =, \geq, >\}$. We define also the set of urgency types $U = \{\epsilon = eager, \delta = delayable, \lambda = lazy\}$. The urgency is associated to each transition in order to define its priority with respect to the progress of *time*, see below

A *timed automaton* over the set of clocks X with actions in the alphabet Σ is a tuple (Q, X, Σ, T, q_0) where Q is a finite set of locations, $T \subseteq Q \times G(X) \times \Sigma \times 2^X \times U \times Q$ is a finite set of transitions and $q_0 \in Q$ is the initial location.

A *configuration* of a timed automaton is a tuple (q, v) where $q \in Q$ is a location and $v \in \mathbb{R}^X$ is a clock valuation. The automaton moves from one configuration to another by performing either discrete or timed transitions. Discrete transitions are of the form $(q, v) \xrightarrow{a} (q', v')$ where $a \in \Sigma$ and there is a transition (q, g, a, r, u, q') such that v satisfy the guard g and v' is obtained by resetting to zero all clocks in r and leaving the others unchanged. Timed transitions are of the form $(q, v) \xrightarrow{t} (q, v + t)$ where $t \in \mathbb{R}, t > 0$ and there is no transition (q, g, r, u, q') such that: either $u = \delta$ (delayable), v satisfy g, and $v + t$ does not satisfy g; or $u = \epsilon$ (eager) and v satisfy g. Intuitively, *eager* transitions must be executed as soon as they are enabled and waiting is not allowed; *lazy* transitions are never urgent, that is, when a lazy transition is enabled the transition may be executed or, alternatively, the process may wait without any restriction; finally, when a *delayable* transition is enabled, waiting is allowed as long as time progress does not disable it.

A configuration (q, v) of the timed automaton is called accepting if the location q is accepting. We define the *real-time language* $L(A)$ accepted by the automaton A as being the set of finite-length traces $l_0, l_1, ...l_n$ from $(\Sigma \cup \mathbb{R})^*$ leading from the initial configuration $(q_0, \{0\}^X)$ to some accepting configuration.

2.2 Decidability results

Timed automata are particularly interesting for verification. In fact, the reachability problem i.e, deciding if a control location is reachable or not, is known to be decidable (Alur and Dill 1994). Moreover, the emptiness test, deciding if a real-time language is empty or not is also decidable. Furthermore, real-time languages are closed under intersection.

Unfortunately, not all the properties of finite-state automata and finite-state languages remain true for timed automata and real-time languages. For instance, real time languages are not closed under complementation. In turn, the language containement problem, that is, deciding if a real-time language is contained in another one, is undecidable in general. For a recent survey of the negative results on timed automata see (Tripakis 2003).

Nevertheless, we can solve some of the above problems with some restrictions. In particular, deterministic timed-automata can be effectively complemented. More precisely, given timed automaton A we can effectively compute the $\overline{A}$ automaton such that $L(\overline{A}) = \overline{L(A)}$. The construction is the same as for finite state automata, see for example (Aho *et al.* 1986). If A is deterministic, the language containment

$L(B) \subseteq L(A)$ is also reduced to an emptiness test, using the classical definition:

$$L(B) \subseteq L(A) \;\Leftrightarrow\; L(B) \cap L(\overline{A}) = \emptyset$$

2.3 Tools

The verification tools we used during this experiment are connected through the IF *validation toolbox* (Bozga *et al.* 2002) developed at VERIMAG. This environment relies on a general *intermediate language* for timed systems, the IF *language*. IF uses timed automata with urgencies as semantic model, but extend them with many other *discrete* features such as discrete variables, communication primitives, dynamic creation and destruction, parameterisation, executable code integration etc.

The IF environment integrates several verification components. It provides static analysis tools allowing for the simplification and optimisation of specifications prior to the verification. Moreover, it provides an interactive debugger as well as an exhaustive model-extractor and model-checker. Furthermore, components allowing for test generation, code generation, scheduling, etc., are also available but they have not been used in this case study.

3 The K9 Rover Executive

The NASA Ames K9 rover 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. In this section we present a description of the system, including a description of what plans look like. In the next section, we describe how for each plan a timed automaton can be automatically generated, which contains all the good behavior of the executive when executing the plan.

3.1 System Description

The Rover executive is a software prototype written by researchers at NASA Ames. It is a multi-threaded system (8,000 lines of Java code), made up of a main coordinating component named Executive, components for monitoring the state conditions ExecCondChecker, and temporal conditions ExecTimerChecker - each further decomposed into threads- and finally an ActionExecution thread that is responsible for issuing the commands to the Rover. Synchronization between these threads is performed through mutex and condition variables.

A plan is a hierarchical structure of actions that the rover 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 a short presentation of the language used in the description of the plans that the rover executive must execute.

The definition of correctness of the rover implementation is that plans are executed correctly. That is, given a plan, the rover shall execute that plan according to its intended semantics.

3.2 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 1 shows the grammar for the language; 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.

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

Figure 1: The concrete grammar of plans.

3.3 Plan Semantics

The nodes must be executed according to the informal algorithm given below. This algorithm has been taken from (Artho *et al.* 2003):

1. wait until the timed part in the start and wait-for conditions are satisfied;

2. if the discrete part of wait-for conditions is not empty, wait for it to become satisfied;
3. check the discrete part of the start conditions: the node fails if it is not satisfied, the execution continue otherwise;
4. put in background a thread to check the maintain conditions (both timed and discrete) if they are not empty; these conditions are checked on every memory update: if they ever becomes satisfied the node fails;
5. put in background a thread to check the timed part of the end conditions; if the time exceeds the end condition, the node fails;
6. the remainder of the execution is type-specific i.e, depends if the node is a task or a block: for task nodes, the action is performed, for block nodes, the inner nodes are executed sequentially, according to their given order in the block node list;
7. finally, if the end of execution arrives inside the timed part of end condition the node successfully terminates; otherwise it fails.

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.

4 Verification

The correctness requirement states that the executive execute the plans according to their semantics. In other words, for any plan, any possible execution must conform to the semantics of the plan. We can not verify this requirement in general, however, we can test it for any plan apriori fixed. More precisely, the verification proceeds through the following steps:

• *define a formal semantics for plan executions*

The meaning of plan executions is described informally in (Artho *et al.* 2003). Here, we give a constructive semantics using timed automata. The timed automaton encodes all the *good* executions of the plan. Moreover, it is deterministics by construction and therefore it can be complemented. The complemented automaton, called hereafter *plan observer*, encodes all the *bad* executions for the plan, and will be run in parallel with the executive (see later) in order to detect execution errors.

• *build an abstract executable formal model of the K9 Rover*.

The K9 Rover executive is a highly non-deterministic software: it is designed as parallel composition of threads and runs in an open environment. For a fixed plan, it is therefore possible to obtain several executions, depending on the interleaving of threads and on the external environment. Since we want to check that *all* executions conforms

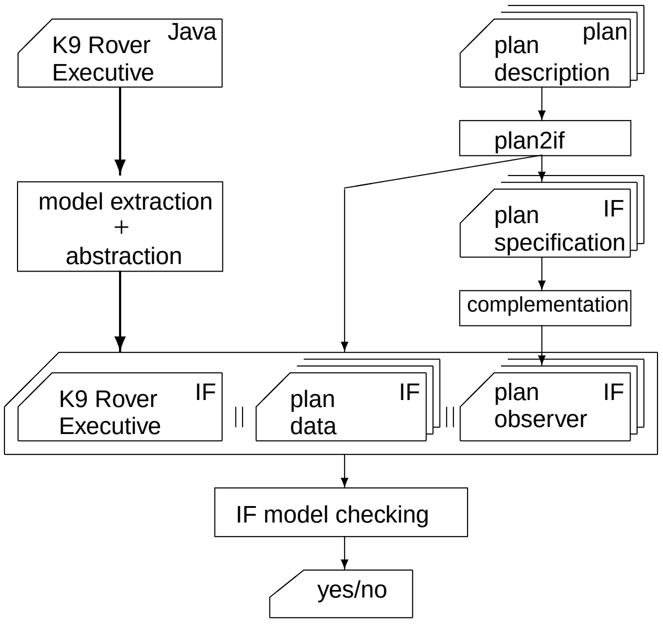

Figure 2: Our approach to the verification problem

to the plan observer, we need a way to enumerate all of them, in a finite amount of time. That is, we need an abstract executable model of the K9 Rover. This model has been extracted from the Java source code as a parallel composition of extended finite-state automata.

If the completeness is not an issue, that is, we are not interested to verify the correctness, but only to test it, this step may be skipped. Instead of checking that all the possible executions are conform, we can restrict us to a finite number of them. That is, just run the K9 Rover several times and check only the obtained executions.

• *check the conformance of the executive*.

This step simply consists of performing the synchronous execution of the abstract executive on some plan and the corresponding plan observer. If error states are reached by the observer, the executive is not conform i.e, there are bad executions produced by the executive on that plan. Otherwise, the executive is conform for the given plan.

4.1 Model extraction

Executive

The first step in the verification process has been the construction of an abstract model of the (actual) implementation of the K9 Rover executive. There are currently automated tools allowing to extract such models directly from the code, however, they are usually limited by the size of the code and its inherent complexity. Instead, we perform the extraction of the abstract model by hand, still following a quite systematic approach.

The general architecture of the Rover implementation is shown in figure 3. In order to keep the abstract model with

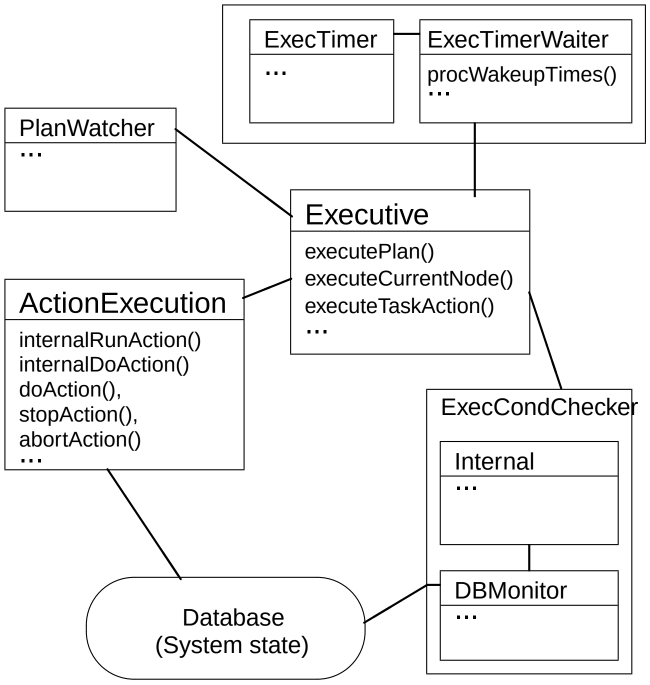

Figure 3: The K9 Rover Architecture

manageable sizes, not all the components have been considered. We focus on the critical parts concerning the plan execution i.e, the Executive, ActionExecution and Timer threads whereas the non-critical parts such as the Database or the execConditionChecker have been abstracted away. We assumed that they work properly.

We treat the execution of a single plan apriori fixed. The plan is stored in a particular data structure containing all the useful information (timing constraints, failure control, etc). The execution of the plan is controlled by the Executive thread. It mainly goes through the nodes in depth-first order and, for each task node reached, it triggers the execute method in ActionExecution thread. When execute gets called on ActionExecution, first, it aborts the currently running task if not yet terminated, then, it installs and launches the new task. On termination, it signals back the Executive thread. The concrete execution of tasks is external to the model.

Timing plays a crucial role in the plan execution. Two special threads, ExecTimer and ExecTimerWaiter, are used to handle the set of the timing constraints attached to the active nodes. Intuitively, the Executive forwards to ExecTimer all the constraints reached (start, wait-for, maintain, end). The ExecTimer keeps the list of incoming constraints ordered by the (absolute) expiration time. When the time reaches the first expiration date, the constraint is removed and the ExecTimerWaiter informs the Executive accordingly.

Plans

The second step in the verification process has been the formalisation of plans and their intended (ideal) execution. 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 simplicity, we consider the following abstract syntax for plans.

Definition 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: $\forall n_1, n_2 \in N, n_1 \neq n_2.\ \hat{\delta}^+(\{n_1\}) \cap \hat{\delta}^+(\{n_2\}) = \emptyset$*
- *$\lambda : N \to \mathcal{A} \times \mathcal{I}^4 \times \mathcal{B}$ is the node labeling function, where $\mathcal{A}$ 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*

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 us first introduce some notations for some given plan $P = (N, \delta, \lambda, n_0)$. The set of actions A_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 $c^\bullet$ the constraint $[-\infty, u]$ where the lower bound of c has been removed.

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

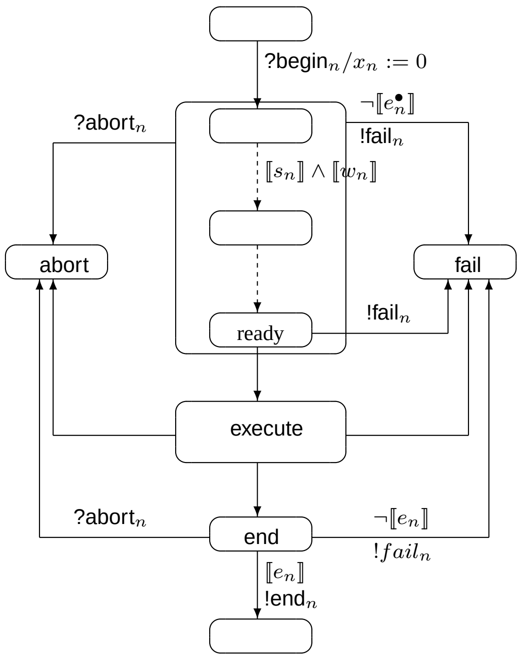

Figure 4: Timed automaton defined for common part.

Definition 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 5. The specific part is filled according to node attributes as shown in figure 5.

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

4.2 Results

The plan translation algorithm has been effectively implemented. The translator takes a concrete plan description and produces the corresponding network of timed automata. The product timed automaton is computed statically, then complemented in order to obtain the timed plan observer.

We experiment with plans of different sizes, having different timing constraints or failure control schemes. Several simple examples are listed in appendix. Our model of the executive performed always correctly, that is, for a given plan, all possible traces produced by the executive on that plan are accepted by the corresponding timed observer.

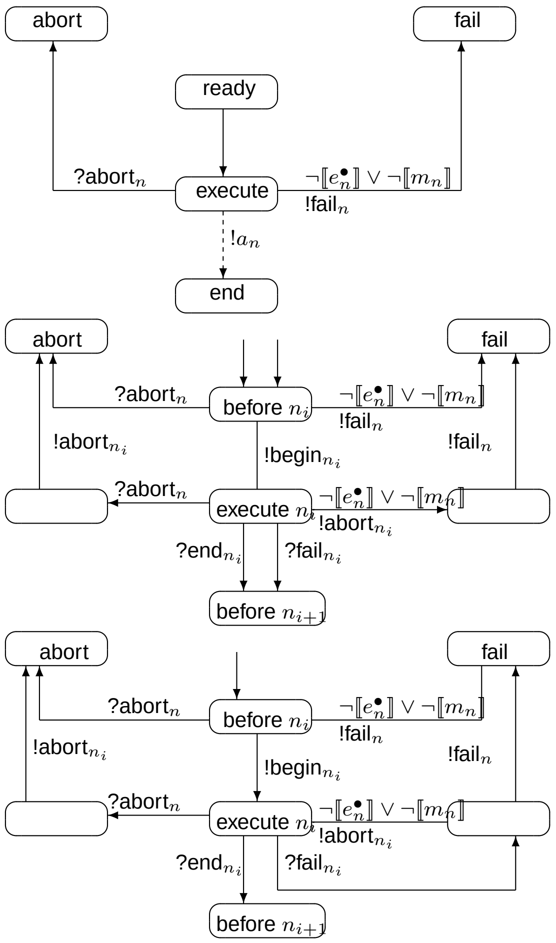

Figure 5: Timed automata defined for respectively the task specific part (top), block specific part with continue on failure on (middle) and block specific with continue on failure off (bottom).

The table 1 gives some details about the size of the models obtained on plans given in the appendix.

5 Discussion

The most difficult part of this experiment has been the construction of an abstract specification for the Rover executive. This specification has been extracted manually from the Java code. In consequence, despite the usage of a systematic translation approach, it may not reflect exactly the actual code. To avoid this problem, one possibility is to use automatic model extraction tools such as Bandera (Corbett *et al.* 2000). Nevertheless, the models have to be anyway extended with timing information about the execution platform. This information is not present in the source code, but is mandatory for the verification of timing related properties.

Another possibility is to focus on *testing* rather than verification, that is finding bugs rather than proving the software to be correct. The idea is to check, for some fixed plan, the

		plan model		*executive model*	
plan	*size*	*states*	*trans.*	*states*	*trans.*
A	3 nodes	127	179	323	459
B	3 nodes	196	263	771	1111
C	6 nodes	164391	231952	3269	4800
D	6 nodes	1617	2286	4172	6184

Table 1: The *plan model* columns (states + transitions) gives the size of the model of the network of timed automaton for the given plan.The *executive model* columns (states + transitions) gives the size of the model of the executive when running the given plan.

traces produced by the executive against the corresponding plan observer. Common traces denote bad executions, that are, bugs in the software. The big advantage is that there is no more need for an abstract model of the software i.e, plans are enough to define the correctness requirement. But, the results are always partial because we will check for a subset of executions and not for all of them. This approach have been investigated in (Bensalem *et al.* 2004) and the results obtained are very encouranging.

Furthermore, in this work we did not consider the verification of plans themselves. Each plan has been taken *as is* and transformed into a requirement for the executive sofware. Nevertheless, checking plan consistency is another problem which can be handled using our methodology. In fact, the timed automata model captures precisely the semantics of plans and can be therefore used to model check any relevant properties. Such properties include for instance feasibility (allow successfull execution), absence of timelocks or deadlocks, response times, etc.

Finally, another interesting perspective concerns the extension of these results to *parameterized* plans. That is, instead of considering individual plans where delays are bounded by integer values (e.g, in the interval $[2, 4]$) consider parameterized plans where delays are expressed with respect to symbolic parameters (e.g, in an interval $[a, 2a]$, with $a \geq 2$). Such a parameterized plan captures a potentially infinite family of plans, each one being obtained by some valid instantiation of parameters with integer values. Parameterized plans still can be compiled to a parameterized timed automata model and can be directly analyzed using specific techniques such as (Annichini *et al.* 2000).

References

A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Readings, MA, 1986.

R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Proceedings of CAV 2000*, volume 1855, pages 419–434. Springer, 2000.

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 *Proceedings of 10th International Workshop on Abstract State Machines, ASM 2003*, March 2003.

Saddek Bensalem, Marius Bozga, Moez Krichen, and Stavros Tripakis. Testing conformance of real-time software by automatic generaton of observers. In *Proceedings of Runtime Verification Workshop, RV'04*, April 2004.

S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *International Symposium: Compositionality - The Significant Difference (Holstein, Germany)*, volume 1536 of *LNCS*. Springer, September 1997.

M. Bozga, S. Graf, and L. Mounier. If-2.0: A validation environment for component-based real-time systems. In K.G. Larsen Ed Brinksma, editor, *Proceedings of CAV'02 (Copenhagen, Denmark)*, volume 2404 of *LNCS*, pages 343–348. Springer, July 2002.

Guillaume Brat, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, and Willem Visser. Experimental evaluation of v&v tools on martian rover software. In *SEI Software Model Checking Workshop*, 2003.

J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.

J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

R.W. Floyd. Assigning meanings to programs. In *In. Proc. Symp. on Appl. Math. 19*, pages 19–32. American Mathematical Society, 1967.

B.S.L. Hantler and J.C. King. An introduction to proving the correcteness of programs. *ACM Computing Surveys*, 8(3):331–353, 1976.

K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), April 2000.

Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, 1991.

K.L. McMillan. *Symbolic Model Checking: an Approach to the State Explosion Problem*. Kluwer Academic Publisher, 1993.

S. Tripakis. Folk theorems on determinization and minimization of timed automata. In *Proceedings of First In-*

ternational Workshop on Formal Modeling and Analysis of Timed Systems, FORMATS'03, September 2003.

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

A Examples of plans

A.1 Plan A

```
(block :id a0
  :start-condition (time 0 2)
  :end-condition (time 2 4)
  :continue-on-failure f :node-list (
    (block :id a1
      :start-condition (time 1 3)
      :end-condition (time 1 4)
      :continue-on-failure t :node-list (
        (task :id a2
          :start-condition (time 1 6)
          :end-condition (time 1 8)
          :action a2)))))
```

A.2 Plan B

```
(block :id b0
  :start-condition (time 0 2)
  :end-condition (time 2 4)
  :continue-on-failure f :node-list (
    (task :id b1
      :start-condition (time 1 3)
      :end-condition (time 1 4)
      :action b1)
    (task :id b2
      :start-condition (time 1 6)
      :end-condition (time 1 8)
      :action b2)))
```

A.3 Plan C

```
(block :id c0
  :start-condition (time 0 2)
  :end-condition (time 1 8)
  :continue-on-failure t :node-list (
    (block :id c1
      :start-condition (time 1 3)
      :end-condition (time 1 7)
      :continue-on-failure f :node-list (
        (task :id c3
          :start-condition (time 2 5)
          :end-condition (time 2 7)
          :action c3)
        (task :id c4
          :start-condition (time 2 5)
          :end-condition (time 2 6)
          :action c4)
        (task :id c5
          :start-condition (time 2 5)
          :end-condition (time 2 6)
          :action c5)))
      (task :id c2
        :start-condition (time 1 6)
        :end-condition (time 2 7)
        :action c2)))
```

A.4 Plan D

```
(block :id d0
  :start-condition (time 0 2)
  :end-condition (time 1 10)
  :continue-on-failure t :node-list (
    (block :id d1
      :start-condition (time 1 3)
      :end-condition (time 1 7)
      :continue-on-failure f :node-list (
        (task :id d3
          :start-condition (time 1 5)
          :end-condition (time 0 1)
          :action d3)
        (task :id d4
          :start-condition (time 3 8)
          :end-condition (time 4 6)
          :action d4)
        (task :id d5
          :start-condition (time 2 6)
          :end-condition (time 4 6)
          :action d5)))
      (task :id d2
        :start-condition (time 1 3)
        :end-condition (time 2 7)
        :action d2)))
```